Asynchronous Teams and Tasks in a Message Passing Environment

HAZELWOOD, BENJAMIN

Asynchronous Teams and Tasks in a Message Passing Environment

Benjamin Hazelwood

A Thesis presented for the degree of
Master of Science (By Thesis)

Department of Computer Science
University of Durham
England

October 2018

Asynchronous Teams and Tasks in a Message Passing Environment

Benjamin Hazelwood

Submitted for the degree of Master of Science (By Thesis)
November 2018

Abstract

As the discipline of scientific computing grows, so too does the "skills gap" between the increasingly complex scientific applications and the efficient algorithms required. Increasing demand for computational power on the march towards exascale requires innovative approaches. Closing the skills gap avoids the many pitfalls that lead to poor utilisation of resources and wasted investment. This thesis tackles two challenges: asynchronous algorithms for parallel computing and fault tolerance. First I present a novel asynchronous task invocation methodology for Discontinuous Galerkin codes called enclave tasking. The approach modifies the parallel ordering of tasks that allows for efficient scaling on dynamic meshes up to 756 cores. It ensures high levels of concurrency and intermixes tasks of different computational properties. Critical tasks along domain boundaries are prioritised for an overlap of computation and communication. The second contribution is the teaMPI library, forming teams of MPI processes exchanging consistency data through an asynchronous "heartbeat". In contrast to previous approaches, teaMPI operates fully asynchronously with reduced overhead. It is also capable of detecting individually slow or failing ranks and inconsistent data among replicas. Finally I provide an outlook into how asynchronous teams using enclave tasking can be combined into an advanced team-based diffusive load balancing scheme. Both concepts are integrated into and contribute towards the ExaHyPE project, a next generation code that solves hyperbolic equation systems on dynamically adaptive cartesian grids.

Declaration

The work in this thesis is based on research carried out at Durham University, the Department of Computer Science, England. No part of this thesis has been submitted elsewhere for any other degree or qualification and it is all my own work unless referenced to the contrary in the text.

The work on enclave tasking in Chapter 2 was recently published as a preprint on a collaborative piece of work [13]. Clarifications on my own contributions are available in the text.

Acknowledgements

First and foremost I thank my supervisor Tobias Weinzierl, for providing the opportunity to carry out this research and the valuable mentoring throughout. Also thank you to Dominic Charrier, for guiding me through the features and complexities of the ExaHyPE codebase. I also give thanks to all members of the ExaHyPE consortium who made this research possible. In particular Leonard Rannabauer for providing the seismic user application from the ExaHyPE project in Chapter 3.

I appreciate support received from the European Unions Horizon 2020 research and innovation programme under grant agreement No 671698 (ExaHyPE) [2]. This work made use of the facilities of the Hamilton HPC Service of Durham University, with thanks to Henk Slim for valuable support on these matters. I furthermore gratefully acknowledge the Gauss Centre for Supercomputing e.V. (`www.gauss-centre.eu`) for funding this project by providing computing time on the GCS Supercomputer SuperMUC and CoolMUC3 at the Leibniz Supercomputing Centre (`www.lrz.de`).

My final thanks go to my family and friends, who provided never-ending support throughout.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

It has often been suggested that the "traditional" scientific method of theory and experimentation has to be extended to include a *third* pillar of science: simulation [46]. Simulations allow scientists to validate theories that either aren't experimentally feasible or for a fraction of the cost. It has allowed for gigantic leaps forward in science and the community surrounding High Performance Computing (HPC) became a thriving research area in its own right. More realistic models require ever larger machines. As soon as the first petascale capable machine went into production in 2008 [4], HPC groups set their sights on the next milestone: exascale. It became clear that previous guarantees about increasing performance, such as simply adding more CPU cores with increasing frequency following Moore's law no longer held at such scales [54]. To progress further, a combined effort in both hardware and software development was needed. In 2011 a large consortium of leading scientists in the

Figure 1.1: A simple visualisation of the 2D Euler equations with the logo of the ExaHyPE project as initial conditions.

field came together to propose the International Exascale Software Project [18]. The extensive roadmap aims to coordinate the advancement towards exascale computing and outlines both the current state-of-the-art and issues surrounding it, alongside potential avenues for future research. This thesis presents two novel contributions to tackle two challenges faced at exascale as identified by the roadmap: faulty machines and synchronisation. A machine is classified as faulty when it does not operate as expected. This can be anything from power outages to performance regressions. The number of machines required for exascale dramatically increases the chance that one will fail, meaning that applications must become tolerant to such faults. With large numbers of machines also come large variations in performance, meaning that synchronisation among machines must be avoided at all cost. It is a waste of time and energy for one process to hold all others back. Algorithms previously designed for tightly coupled execution must be redesigned to allow for performance variations and interruptions.

To provide a context for these issues and the contributions presented in this thesis I rely on the ExaHyPE project. ExaHyPE is a simulation engine designed by an international consortium of scientists to solve hyperbolic equation systems based on highly accurate ADER-DG [21] coupled to robust Finite Volumes. Work on the project begun in 2014 and is anticipated to finish in 2019, with the simulation of "grand challenges" in astrophysics and seismology. The "engine" terminology comes from its design prioritising a separation of concerns between the scientific computing infrastructure and the application developer's code. This means that a group with limited HPC experience but in-depth subject knowledge will be able to solve large-scale problems in a much shorter time-frame than previously possible.

The engine uses arbitrarily high-order Discontinuous Galerkin (DG) techniques that have achieved widespread success as partial differential equation (PDEs) solvers on supercomputers. Their properties allow for extremely high computational efficiency by combining high arithmetic intensity with blocked cache efficient realisa-

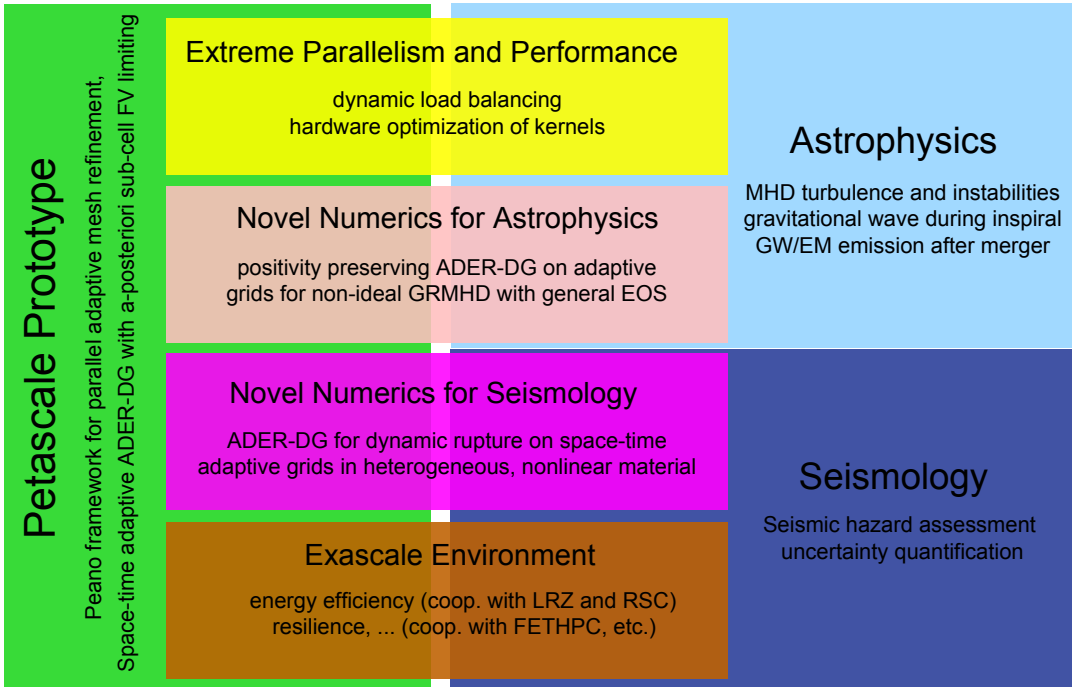


Figure 1.2: An overview of the ExaHyPE project split into work packages. This thesis presents work on the "Exascale Environment" in addition to "Extreme Parallelism and Performance." From the ExaHyPE project proposal.

tions. They are also a great fit for application developers with support for dynamically adaptive block-structured grids [20]. These fit into common predictions about exascale capable codes [19]. Charrier and Weinzierl show that communication can be minimised with ADER timestepping in combination with DG [12]. Recently, Dumbser et al proved in practise that without the overhead of dynamic grids ADER-DG is capable of efficiently scaling up to 180,000 cores on the Hazel Hen supercomputer of HLRS in Stuttgart, Germany [22]. With the ExaHyPE engine, the goal is to achieve comparable results while providing users with the complex grid topologies and user-friendly features. Tavelli et al detail an application for these features in their recent work. They model linear elastic wave equations with complex topographies on adaptive cartesian grids [72]. Figure 1.1 shows an example simulation of the Euler equations.

To achieve its ambitious goals, the project is split into work packages assigned to specialist teams (Figure 1.2). My Chapter 2 contributes dynamic load balancing

to "Extreme Parallelism and Performance." I outline and validate an algorithmic idea called *enclave tasking* developed in collaboration with my supervisor Tobias Weinzierl and colleague Dominic E. Charrier [13]. This approach uses an asynchronous task-based parallelisation paradigm to overlap computation and communication on dynamically adaptive grids.

My Chapter 3 contributes resiliency features to the "Exascale Environment" package. I introduce the teaMPI library, which allows existing applications to form replicated distributed teams from the available resources with minimal changes to existing code [38]. Replication is a previously overlooked form of fault tolerance in HPC attributable to the expensive overhead in resources and performance. With the teaMPI library I show that performance concerns can be vastly reduced from previous implementations with intelligent consistency checks. The use of teams allows the library to detect heterogeneous performance and faults within the individual processes without the significant overhead of previous solutions.

Chapter 4 takes these two contributions and outlines how they may be combined into a diffusive load-balancing approach. Work is shared among teams to reduce the overhead of resources required by replication. I finally give some concluding thoughts on the ideas presented in this thesis.

Chapter 2

Asynchronous Tasks

Note: This chapter is based on a collaborative preprint paper with my colleague Dominic E. Charrier and supervisor Tobias Weinzierl [13]. I originally identified the issues of excessive synchronisation and limited concurrency of the previous approach on adaptive grids. I then proposed the asynchronous tasks which has developed into the work presented in the paper and this chapter. I furthermore carried out the results for the paper and I reuse the data for this thesis.

The development of exascale capable software has proved to be remarkably difficult. Challenges lie in predicting the capabilities of future hardware. For the Peano framework, upon which ExaHyPE is built, this has been visible in the approach to shared-memory parallelisation. When it looked like cache-oblivious algorithms and careful memory usage would be key, the code focused on its strengths of dynamic, adaptive meshes. However, the next claim by hardware and software vendors alike was that the efficient utilisation of many-core processors with wide vector registers would be paramount. Peano was then modified to extract regular subgrids into plain data-structures which could fit into highly vectorised parallel-for constructs. Meanwhile, the HPC community was experiencing widespread success with task-based parallelisation as a means to avoid the explicit synchronisation of parallel-for methods [65]. Charrier and Weinzierl outline how the ADER-DG predictor-corrector

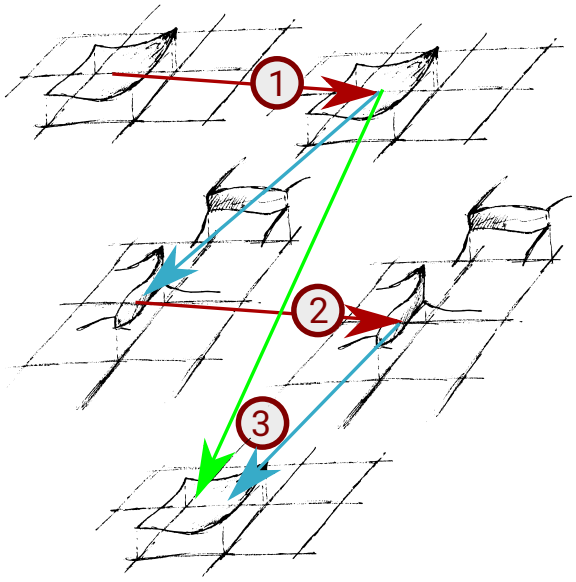


Figure 2.1: A sketch of the handkerchief analogy for the three phases of ADER-DG: predictor, Riemann and corrector.

scheme [21] underlying ExaHyPE can be modelled by tasks and additionally reduce the synchronising communication among processes [12]. Unlike many codes that use a task paradigm, it is neither feasible nor necessary to construct the task graph a priori [65]. This is attributable to the grid data structures which can be dynamically refined per time step.

2.1 ADER-DG In ExaHyPE

To provide the context and motivation for the contributions in this chapter, I summarise ADER-DG for readers unfamiliar with the scheme. Full details are available in the original work by Dumser and Käser [21]. I also outline some ExaHyPE specific implementation details with respect to current parallel approaches.

There are three phases of ADER-DG: prediction, Riemann and correction. Each phase requires the result from the previous, forming dependencies. An apt analogy is the simulation of dropping a handkerchief (Figure 2.1). Instead of running the simulation on the handkerchief as a single entity, it is split into smaller pieces (cells). Moreover, the simulation timespan is split into timesteps. For each piece over a

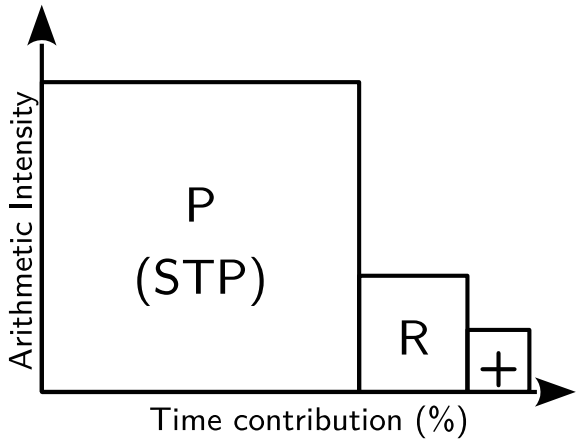


Figure 2.2: Visualisation of the arithmetic intensity and time contribution characteristics of the three execution phases. The '+' label corresponds to the corrector that combines the contributions of the P and R tasks into the next solution. The P tasks in ExaHyPE correspond to the STP.

timestep, the predictor is able to carry out a localised simulation (it locally spans the full space-time polynomial). However, this amplifies any jumps in the solution, such as a gust of wind, resulting in large gaps between the boundaries of each piece. Therefore, a Riemann solve is performed on these interfaces which yields Riemann solutions. To calculate the final state of each piece, the corrector phase sums up the results of the previous phases. Once performed for each piece, the state of the whole handkerchief is known for that timestep. The process can then be repeated for the duration of the simulation timespan.

In ExaHyPE the handkerchief represents the computational domain, split into cells that are traversed by the PDE framework Peano [77]. The three phases per cell are modelled as tasks. The prediction task P, is the local higher order integration of the PDE per cell. These P tasks typically take up the majority (75%+) of the execution time. It has no dependencies from neighbouring cells. The result of this task is then passed to a Riemann solver that is modelled as a second task, R. The Riemann solver takes in data from the faces (interfaces) of neighbouring cells, forming dependencies among tasks. Finally, the outcome of the two tasks is fed into the corrector task +, to calculate the next solution.

To utilise the performance of modern machines, the compute characteristics of each task is important. The higher-order integration P of the PDE is compute-intensive provided a high enough polynomial order p and a complex PDE. Typical ExaHyPE applications use $3 \leq p \leq 9$. It is also extremely cache-efficient as the data is located in small array blocks. On the other hand, R tasks feature an extremely low arithmetic intensity and are highly memory-bound. Each issues few flops per byte that must be loaded from main memory. The corrector is usually cheap to compute, requiring few calculations. However, if neighbouring cells are of differing size (refinement levels), or require limiting with a Finite-Volume scheme, it can be complex. Any parallel ordering of the tasks must not only satisfy the dependencies but also account for these compute characteristics. If possible, the R and P tasks should be intermixed among cores to avoid saturating the memory resources available. If all R tasks are ran concurrently it will place excessive pressure on the memory subsystem. A summary of the algorithm phases and their contributions to the runtime is given in Figure 2.2.

As previously hinted if the grid were regular or even statically refined this would be relatively trivial to manage. The task graph with its dependencies and properties may be passed to a scheduler such as METIS to produce a satisfactory ordering for the duration of the applications runtime [45]. An example application using this approach is the SWIFT code designed by Schaller et al [65]. However, with dynamic adaptivity the task graph becomes much more complex as the prolongation and restriction along resolution boundaries has to be done in a well defined order. Task dependencies also change along with the grid. This would render any task graph invalid. Concurrency is difficult to exploit in these regions of the grid. In ExaHyPE, the grid may change upon every traversal as few constraints are made about the adaptivity.

To facilitate distributed-memory parallelism, Peano splits the computational grid into non-overlapping subdomains. Each process then owns one of these subgrids,

which increases the complexity of the dependencies further. Cells along the boundaries must send and receive face data from neighbouring cells on other processes. The parallel ordering must be entirely deterministic.

2.2 Summary Of Contributions

The contributions of this chapter are a novel task invocation paradigm, enclave tasking, which intelligently schedules work throughout the mesh traversal. Localised regions of tasks are processed dependency-free from a background queue. The implicit task graph yields high levels of concurrency. As the Riemann problems are computationally cheap and must not be grouped into bursts for maximum throughput, these are embedded directly into the traversal whenever the face is loaded. This ensures that the bursts are optimally spread out. During the traversal the computationally heavy P tasks are placed into a queue that is shared among threads. Tasks are only polled for completion at the end of the traversal. Throughout, threads are free to dynamically steal work from this queue to ensure maximum concurrency and intermix the R and P tasks. The producer-consumer idiom is well-suited to this algorithm.

However, once the dynamic adaptivity and communication cells are considered, it is no longer possible to spawn all P tasks into the shared queue. In both cases the execution ordering must be well-defined and deterministic, yet there still exists large areas of the grid where such constraints are not imposed. If the ratio of such cells to the rest is low, it is enough to processes them during the traversal in addition to the R tasks. Provided that the other threads remain busy with the work in the shared queue then the concurrency will not be effected. To support such a scheme, a marker-and-cell realisation is implemented. The cells on refinement and sub-domain boundaries are marked and processed during the traversal. When visualised on top of the grid, the marked cells form a *skeleton mesh* around large areas of trouble-free

cells. The term used to define such areas is *enclaves*, first used in a HPC context by Sundar and Ghattas [71].

The second contribution of this chapter is owing to an additional benefit that the enclave concept provides. By postponing the execution of the computationally heavy P tasks, the cells along the MPI boundaries get processed much earlier during the traversal. This increases the length of the communication window as the data may be sent out earlier. Communication may be overlapped with computation. Waiting for incoming data is wasting time and energy. A naïve implementation with only non-blocking communication function in MPI does not immediately lead to asynchronous transfers of data. To achieve this the MPI library must be regularly polled to ensure the "progress" of messages in the network subsystem. Most applications therefore accept that they must sacrifice a whole thread per node for this important functionality [39, 75, 79]. However, the producer role in the enclave concept can be exploited to ensure message progress natively. Memory and network bandwidth are critical resources, and it is predicted that these will be the bottlenecks for many exascale HPC applications [18]. Therefore in addition to the properties of enclave tasking, intermixing memory and compute intensive tasks, I also investigate variations on the communication scheme to minimise the contribution of data transfer through the network on the ExaHyPE code. The concept of overlapping communication with computation is not new, especially with DG [3, 47, 71], but I summarise the cumulative advantages and unique properties of the enclave tasking concept:

1. A task schedule is derived from the grid on the fly, allowing for unconstrained dynamic adaptivity. Constructing such a task graph would be expensive if the grid is frequently modified by the adaptivity [48, 50, 66].

2. No assumptions about the grid structure are made, or restrictions to specific subgrid regions/enclaves [47, 66, 71].

3. Tasks of different compute characteristics are efficiently mixed to avoid satu-

rating critical resources.

4. Overlaps communication with computation *without* sacrificing a whole thread for ensuring message progression.

Enclave tasking combines all of these concepts into one powerful methodological tool. The three important phases of the code are all able to operate concurrently: while the lightweight producer supplies tasks to the thread-shared work queue, it kicks off MPI transfers and polls the MPI engine. The threads and network subsystem are then free to complete their tasks completely asynchronously until the barrier at the end of the traversal. Such an approach minimises the idle time of all critical resources: memory, threads and network.

The remainder of the chapter is organised as follows: I describe how the task graph is implicitly constructed from the computational mesh and operator constraints (Section 2.3.1) before discussing the implementation details of the enclave tasking and MPI realisation code in Section 2.3.3. I then present the results of scaling up to 756 cores on an example ExaHyPE benchmark application (Section 2.4). Finally, I present a brief outlook into future work on enclave tasking and the shortcomings of the present approach.

2.3 Implementation

2.3.1 Discontinuous Galerkin On Dynamically Adaptive Cartesian meshes

I now expand on the overview given in Section 2.1, outlining all tasks with dependencies and compute characteristics which must be managed by the proposed task runtime.

As previously discussed, in ExaHyPE a PDE is first evaluated with a higher-order integration over all cells of the computational grid. This Space-Time predictor

(STP) phase contains the P tasks for our enclave tasking approach. However, as DG represents the grid as discontinuous polygons, a Riemann solve is introduced that is then integrated over all faces of the cell. The work of Charrier and Weinzierl cast the whole ADER-DG scheme into tasks [12], and I use a representation of P and R tasks for cell-wise and face-wise integrations here. As both use the same PDE terms, they are directly comparable in terms of arithmetic intensity with a disregard to the cache efficiency [78]. For relatively simple PDE's, an intensity of around 0.1 flop/byte is expected. However, it is more useful to consider the cache-aware roofline model proposed by Ilic et al [41]. This model accounts for the flops performed for data that already resides in main memory. As the integration over the cell with high order polynomials is stored in small array blocks, the arithmetic intensity relative to the caches is much higher than the integration over the faces. These must usually load the data from main-memory. The STP per cell is completely independent of other cells with the discontinuous polygons of DG. As the polynomial order is increased, the P tasks become increasingly compute-bound. Charrier and Weinzierl show that this decomposes into a single independent task per cell, P. This task mandates an output dependency only onto follow up computations.

The integrations over the faces also provide one task, R, per grid face. Although the R tasks of the faces of the cell are independent, each requires input from the adjacent cells from a Riemann solve. This couples cells with shared faces, with $2 \cdot d$ of these dependencies per cell. Provided no resolution boundary exists between the cells, a simple extrapolation is enough or else the outcome is projected from more/less neighbouring cells. The Riemann solves require data usually residing in main memory because of cache capacity. Each contain few floating point operations, and are memory bound.

When talking of tasks and dependencies it is helpful to visualise the task graph (Figure 2.3). Two types of tasks exist, cell P and face R tasks. The P tasks have no spatial dependencies, only temporal upon the previous solution. The face tasks

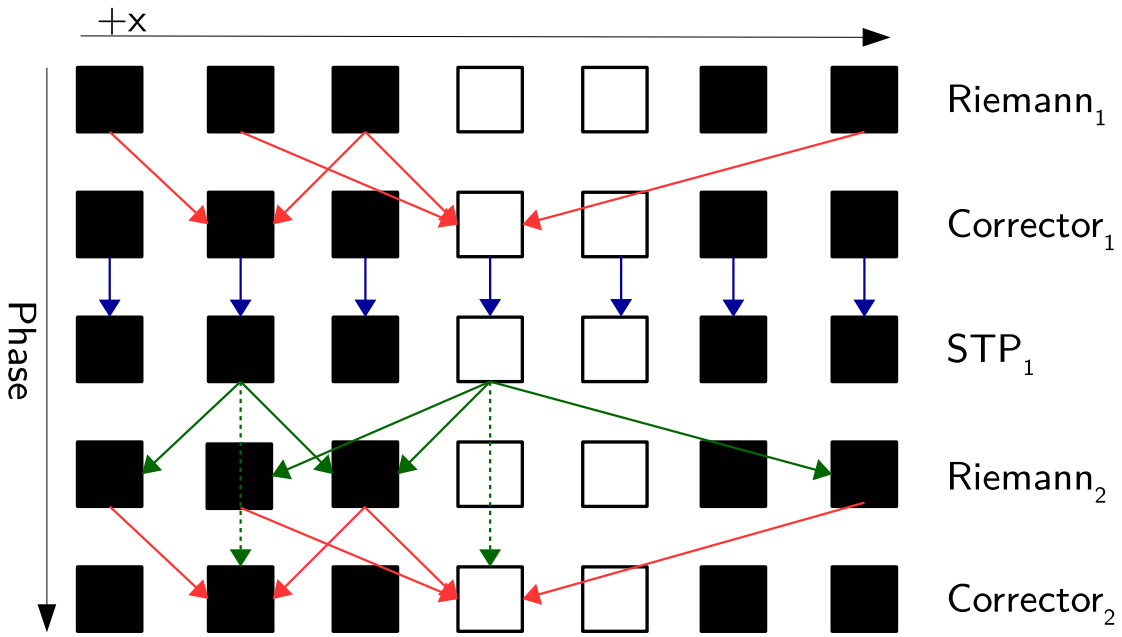


Figure 2.3: A subset of task dependencies for the three phases of computation in a simplified 1D case. Enclave cells are shown as filled. Skeleton cells are empty. The corrector combines the STP and $2 \cdot d$ Riemann solves. Cells along refinement boundaries (empty), may require more or less input faces for the Riemann solves.

from the Riemann solves require input data from all neighbouring cells. More or less cells input data into the R tasks along adaptivity boundaries. The two task types leads to two traversal types: one over the cells, one over the faces. Each traversal type is embarassingly parallel, with no internal spatial dependencies among them. Dependencies only exist between traversal types. All R tasks are independent of other R tasks, and likewise for P tasks.

Two main variations on the traversal exist depending on whether the P tasks feed into the R tasks. If no such dependency exists, then both may be spawned directly in a single traversal and represents the most trivial case. At the end of the traversal (or at the start of the next), there must be a wait barrier for any pending tasks. The next traversal then combines the contributions from the two task types and launches the computation of the next solution. No task graph or complex depending tracking required.

When R tasks depend on P tasks it gets a little more tricky. By sticking to a

single traversal when spawning the R and P the dependencies must be also passed with the tasks. Although this is supported in libraries such as TBB among others, it is more efficient to resolve these dependencies ourselves with domain-specific knowledge. Therefore the following approach is proposed. First, the R tasks should be issued when the face is loaded during the traversal. When the 2^d adjacent faces of a cell have been updated, the P tasks may be spawned. The issue with this approach is that with explicit time stepping it requires such time stepping to be optimistic as shown by Charrier and Weinzierl [12]. Such a scheme is supported in ExaHyPE, where the three phases are fused together into a single timestep. P tasks could then be spawned with an inadmissible time step size. By neglecting the dependencies in an optimistic fashion it may be the case that some computations be rolled back and done again with a correct time step size.

Even with efficient dependency management undertaken, this approach still faces several complications in a real-world implementation. I discuss some of these complexities here and the solutions are presented in the following section.

ExaHyPE implements a non-overlapping domain decomposition. This means that any cells along the boundary contribute only half of the data for the Riemann solves (R tasks). These are computed redundantly on both ranks, which then must swap data with the required neighbours. If the tasks of such cells were passed to the job scheduler then the permutation would not be known a priori, which severely complicates the MPI exchange. It is more efficient therefore to impose temporal dependencies on these cells such that the data can be exchanged deterministically.

The second issue stems from the maturity of task runtime systems such as OpenMP, TBB or C++. Out of the box each struggle to balance the complex requirements for balancing the execution of R and P tasks. For example, all threads must not access the main memory concurrently by avoiding bursts of cache capacity misses.

The Riemann solves often impose additional constraints on the ordering of tasks.

In the work of Berger and Colella they project the solution onto the finest grid along refinement boundaries and solve the problem there [6]. The outcome must then be restricted up to the coarsest grid. If the grid traversal is modelled as a depth-first recursive function this is simple to realise. The restriction operators may be done while backtracking to the coarser grid. With a complex grid this will impose many task dependencies as the computations on the coarser cell must complete before following the recursion down to finer levels and vice versa. This effectively serialises the mesh traversal and may starve the task consuming threads.

The performance of the implementation will be dictated by the throughput of tasks. This will be maximised with an ability to:

1. Avoid saturating the network and memory subsystem while cores remain idle.
2. Avoid synchronisation with other tasks.
3. Exhibit high levels of potential concurrency for many-core processors

Dynamic adaptive mesh refinement typically makes satisfying these constraints challenging. If cells refine late in the traversal, the allocation of memory and initialisation of data structures will have to be completed before eventually spawning the tasks. By this time the task-consuming threads will be starved of work. In the following section I show how enclave tasking is able to avoid such scenarios.

2.3.2 Enclave Tasking

In ExaHyPE, simple constraints are placed upon the computational mesh. The first is the topology of the underlying grid, which is typical of a many software solutions. A user provides the application with a *maximum-mesh-width* and a *maximum-mesh-depth*. Starting with a conformal grid of a single cell, the initial regular mesh is created by dividing the cell a fixed number of times equidistantly until each cell is at most maximum-mesh-width in size. Only square cells are supported. This level is defined at ℓ_{min}. The Peano package implements tri-partitioning, splitting

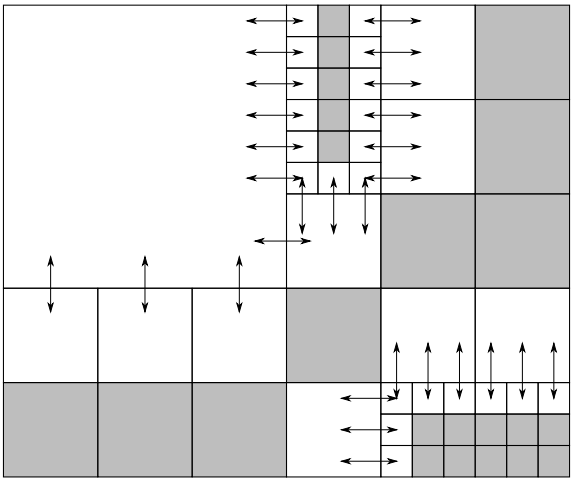


Figure 2.4: An adaptive Cartesian mesh (not distributed among processes) where the Riemann solves along adaptivity boundaries are shown by arrows. Cells involved in Riemann solves form a skeleton around the enclave cells in grey.

a cell results in 3^d cells on the finer level. Topologies of this type are quadtrees, octrees and forests of trees [20]. The work of Berger and Colella also imposes such constraints [6]. In ExaHyPE, tree grids are used.

The user then specifies a refinement criterion, of which there are two types. If the criterion is only specified for time $t = 0$, it follows a static refinement pattern and the grid will remain the same for the duration of the simulation. Although naturally such a set up is supported, it does not make use of the advanced dynamically adaptive mesh refinement capabilities of the ExaHyPE engine. Performance trade-offs have to be made to support these features, yet the enclave tasking approach seeks to minimise this. If a user chooses to refine a cell, either statically or dynamically during runtime, the new cells will be created on the finest level denoted $\ell_{\min+1} = \hat{\ell}$. Further refinement is possible, provided that $\hat{\ell} \leq \ell_{\max}$ (maximum-mesh-depth). The assumption imposed on top of the dynamic refinement is that cells only refine and coarsen along existing refinement boundaries. This means that a cell on level ℓ may be refined if one of the adjacent cells is on a finer level. The assumption is extended to coarsening too. A cell can only coarsen if an adjacent cell is also coarser. Cells surrounded by cells on the same grid level will neither refine nor coarsen.

The assumption on the refinement patterns is reasonable in hyperbolic problems. CFL conditions ensure that any shock waves will not propagate further than a cell per time step. If the CFL condition is violated the computations are rolled-back with a new time step. With elliptic problems, users will specify the regions of interest via an initial refined grid. The mesh will then be developed further throughout the simulation. The errors from these regions are mirrored by the grid, where the finer levels follow the problematic regions. The dynamic refinement criterion can be made to refine an additional level around these areas to fit the assumption. A potential sticking point is applications with highly non-linear equations. The areas of interest may be completely non-localised. However, the ExaHyPE project can still be used by applications with such properties. Domain-specific knowledge can be leveraged in tandem with intelligent refinement criteria to predict such "random" areas of interest and avoid non-local mesh refinement. These refinements appear random to the mesh, not the application.

Definition 2.1: Skeleton Grid

Contains the critical cells that occur on communication or refinement boundaries and therefore must be processed with priority yet yield limited concurrency.

Definition 2.2: Enclave Cell

A cell where all neighbours are on the same level and no neighbour is remote.

In Figure 2.4, I show how the grid structure can be exploited by the proposed enclave tasking method. Suppose an adaptive grid is split into sub-domains to exploit distributed memory parallelisation. The *skeleton grid* is then formed on each sub-domain by those cells either requiring communication with other processes or are hosted on a different level to at least one adjacent cell (Definition 2.1). Exploiting the knowledge that skeletons usually occurs in localised areas, it leaves the remaining cells in the grid to form large *enclaves* (Definition 2.2). These cells produce tasks without dependencies that can be processed by idle threads.

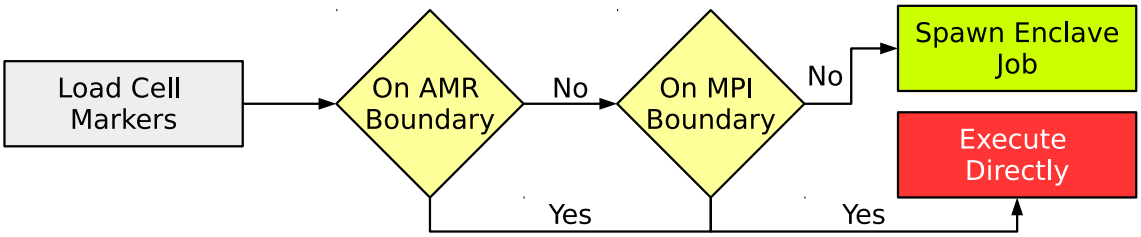


Figure 2.5: A flow chart summarising the decision process made by a producer for a cell during the mesh traversal.

For the mesh traversal scheme outlined in Section 2.3.1, enclave tasking is a task invocation paradigm that decides per cell encountered in line with the traversal:

(a) If the cell is a member of the skeleton grid, process the R and P tasks immediately with the current thread, or

(b) Process the R tasks directly by the current thread and push the P task onto a shared task queue to be processed asynchronously. It may be the case that the same thread picks up this task later when it becomes idle at the end of the traversal with no more "critical" work left.

This choice is summarised by the flow chart in Figure 2.5. At the end of the traversal, any remaining tasks in the queue are waited upon. To increase the rate at which the queue is filled to avoid work-starvation, multiple producer threads traverse the grid. It has no effect on the algorithm described.

Equally, the R tasks could also be placed into the shared queue. However, as they are cheap in terms of computation and exert dependencies on spawning of P tasks, it makes sense to process them immediately during the traversal. This ensures that they are not processed in batches by all threads and thus saturate the memory bandwidth. This insight gives enclave tasking an arithmetic-aware character. Any other small tasks that exist are also processed immediately during the traversal.

I summarise that the described methodology satisfies the various constraints and efficiency concerns of the context described in the previous section. By processing the Riemann solves deterministically directly in the traversal it ensures that no

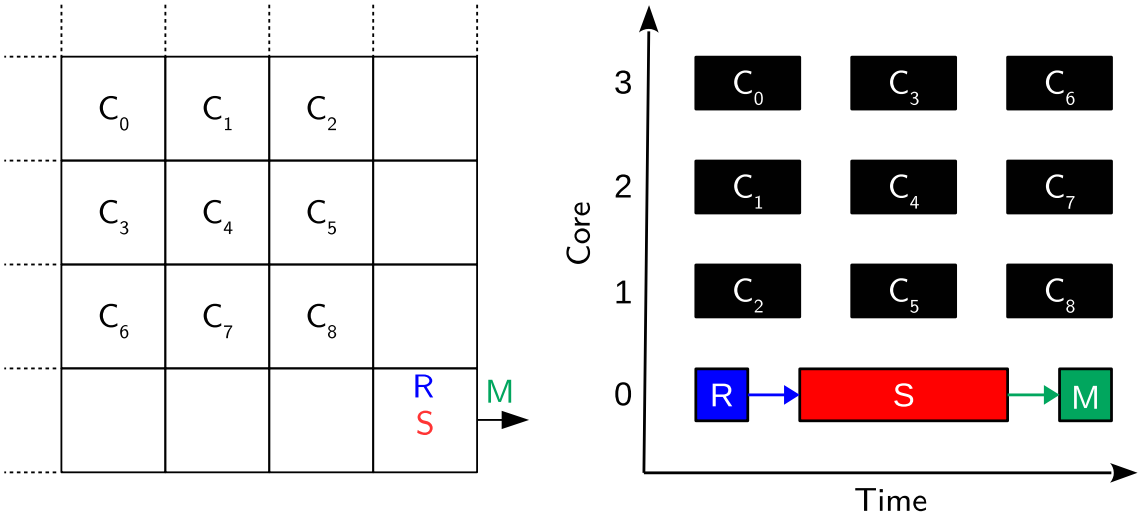


Figure 2.6: An illustrated snapshot of enclave tasking with four cores on a small subdomain of a mesh. Core 0 traverses the grid and processes all Riemann solves (R) and any solves along a refinement transition or communication boundary (S). All remaining cells' P tasks are spawned into a shared queue and are processed when a core runs idle. They take previous R tasks as input dependencies. Tasks from the skeleton cells along communication boundaries trigger message tasks (M). In this case S and R trigger M. Only a subset of skeleton cell's tasks are shown for clarity.

bursts of memory intensive operations occur.

Additionally, this deterministic behaviour is extended to the cells along the non-overlapping domain decomposition boundaries. It is known from the prescribed mesh traversal ordering. As all cells do not immediately process their intensive P tasks, it allows the MPI skeleton cells to be processed earlier. This extends the window allowed for data transfer and later I discuss the implications this has on the MPI data transfer implementation. Is it better to adopt a scheme alike the R tasks and let the data trickle through the network subsystem as and when required or aggregate into one big message that can be sent en masse? I will show that by allowing the data to be sent in the background, the window is large enough that it does not matter in practise.

With the dynamically adaptive refinement also processed deterministically via the producers following the traversal, bursts of memory allocations/deallocations/initialisations do not occur. The Riemann solver algorithms are free to impose any constraints on the mesh traversal ordering without a major negative impact on

the concurrency. While the producer is handling the refinement, the other threads continue to take P tasks spawned by the traversal of earlier cells. An example execution ordering exhibiting the promises of increased concurrency is shown in Figure 2.6.

2.3.3 Tailoring The Task Runtime

So far I have outlined the generic idea of enclave tasking and the benefits it promises. However, there are implementation challenges. At its core enclave tasking relies on a producer-consumer pattern. A main thread(s) follows the mesh traversal and spawns most of the work into a background queue. Any critical tasks are processed directly. Upon completing the traversal, it waits while other threads complete tasks from the queue. To provide some context for the potential pitfalls I first outline an idealistic scenario. While the producer spawns the tasks, the consumers immediately start work on them. With the perfect balancing of workload and hardware resources the traversal will complete and at the same time the threads will have completed their work in the shared queue. Also at this moment, the incoming data will have already arrived from neighbouring processes and then the next timestep can begin. In the rest of this section I explain what improvements to the tasking runtime and MPI communication were made to the original idea to get as close to this situation as possible.

Task Fusion

The first choice is how to manage the potentially millions of tasks created by the traversal. It would be entirely feasible to write an implementation of enclave tasks based on plain threads (pthreads or C++11) and is indeed something we did try at first. However, it soon became clear that it is advantageous to leverage the enormous amount of research-hours put into modern task systems. Although the functional requirements for a dedicated library are small, some are difficult to im-

plement efficiently. The main deal-breaker is the ability to spawn large amounts of asynchronous tasks. Although the popular OpenMP library was considered, the restrictions it places to aid simplicity make it infeasible for a codebase as complex and dynamic as ExaHyPE. In particular, completely asynchronous tasks are not supported, they must be spawned and completed during well defined `omp parallel` clauses. This could change with future versions. Intel's Threading Building Blocks (TBB) proves to be a much better solution [58]. Fully asynchronous "fire-and-forget" tasks are spawned via the `tbb::enqueue` function, which fairly distributes tasks between threads in a roughly FIFO order. Although TBB is theoretically capable of handling the quantity of small tasks, spawning each P task into TBB's own work queue induced significant overhead by polling the task queues too frequently. The small tasks also suffer from NUMA [49]. Intel's own recommendation is that tasks spawned via the `enqueue` command must contain at least 10,000 execution cycles [43]. Therefore, an additional task management layer is added atop TBB to fuse tasks together. To differentiate between real tasks and asynchronous enclave tasks, the latter are referred to hereon as *jobs*.

Now when a task is "spawned" it is actually added to a `tbb::concurrent_queue` shared among threads. The actual TBB tasks that get spawned (via `tbb::enqueue`) are referred to as *job consumers*. With n cores available, up to $n-1$ of these job consumers may be running at a time. During the traversal, it is usually more efficient to allocate resources fairly between producer and consumer threads until the end of the traversal. At this point, all $n-1$ consumers are spawned or are already running. This minimises the traversal time ensuring tasks are not starved of work and that MPI data is sent out early. At the end of the traversal the remaining consumer tasks are then spawned to quickly process the remaining jobs.

The consumer tasks take jobs from the shared queue and process them. Once finished with their tasks, they requeue themselves. Deciding upon the amount of tasks a consumer should take from the queue is a complex decision, and depends

on a variety of factors and constraints. A grain size of tasks must be chosen. For example, if the P tasks for one application are computationally cheaper than another (i.e. fewer PDE terms or lower order), then it should take more tasks in one rush. Machine properties also come into play here, and if too few jobs are taken at once then contention among threads will occur. The literature agrees no single grain size represents a global maximum in throughput. Rather it is better modelled as a plateau with a wide range of acceptable choices [57]. Although this could be deployed to the machine learning model designed by Charrier and Weinzierl [14], in ExaHyPE a simpler deterministic model was chosen (Equation 2.3.3) to balance efficiently fuse tasks for consumers.

$$\max\left(\frac{j}{n}, j_{\min}\right)$$

Where j is an estimate of the number of jobs in the queue, $j_{\min}$ is a fixed value for the smallest number of jobs that may be taken and n is the number of consumer tasks available. Although it may be modified on a per application basis, $j_{\min} = 4$ has proven to be a suitable value for many applications tested with ExaHyPE. This manual job stealing works well in tandem with TBB's own task stealing mechanisms and allows for fusing multiple asynchronous tasks.

2.3.4 Tailoring The MPI Runtime

By prioritising the execution of critical tasks along MPI sub-domains, enclave tasking is implicitly "communication-aware". If the data transfers are initiated with the non-blocking operations (`MPI_Isend/Irecv`) then it is possible to perform the communication asynchronously. Meanwhile, the tasks continue to be executed by the job consumers. Although this is not a feature unique to enclave tasking, other solutions usually require workarounds [3, 47, 71]. It fits elegantly into the methodology outlined here. The work of Hoefler and Lumsdaine dives into these issues and so-

lutions in detail [39]. However, two fundamental concepts underpin successful MPI background communication: the communication protocol and message progression.

Definition 2.3: MPI Message Progression

The act of tripping the MPI engine to ensure pending (non-blocking) operations complete.

Definition 2.4: Eager Protocol

The sender assumes the receiver has enough memory to buffer the message without posting the matching receive.

Definition 2.5: Rendezvous Protocol

The sender must wait until the matching receive has been posted before beginning the data transmission.

MPI implementations are free to choose whatever communication protocols they want provided that it adheres to the MPI standard. Two common protocols used by most implementations are eager and rendezvous (Definition 2.4-2.5). If a message is sent eagerly, the sender assumes the receiver will have enough memory to buffer the message even if the matching receive has not yet been posted. In contrast, the rendezvous protocol requires the matching receive to be posted before the communication can begin. Although eager sending minimises synchronisation between communicating ranks, it may incur significant memory and performance overheads if used excessively with large messages. Therefore, MPI implementations commonly have a message size cut off parameter to switch between the two.

Sometimes the MPI standard is deliberately vague to give the implementations some leeway in (as of yet) unsolved challenges. Automatic message progression is one example [39, 79]. To get non-blocking long-running MPI functions to complete in the background, it is necessary to "progress" the underlying MPI engine to complete the communication. This can be achieved with five common, not necessarily orthogonal, approaches:

1. Call `MPI_Test` on the `MPI_Request` object returned by the non-blocking operation. Multiple function calls may be necessary to complete the request.

2. Call `MPI_Iprobe` on the receiving rank, with source and tag parameters. The source and tag can be their `MPI_ANY_SOURCE` or `MPI_ANY_TAG` wild card values respectively, but this will incur additional overhead. Again multiple function calls may be necessary to complete the request.

3. Call `MPI_Wait` on the request object returned by the non blocking function.

4. Call `MPI_Probe` on the receiving rank, with a source and tag parameters. Again these can be replaced with their wild card counterparts.

5. Use a dedicated helper thread to ensure progress automatically. Often provided by MPI implementations, although some applications implement their own [75].

The first two approaches are semi-asynchronous, as they allow computation to occur between `MPI_Test` or `MPI_Iprobe` function calls. The next two convert the operation to blocking and defeat the point of non-blocking communication. They will not return until a message has been received, ensuring progress. These are usually used when no more work is available and the application has to wait for the communication to complete before proceeding. Although this is the fastest and simplest way to ensure progress, it naturally defeats the point of asynchronous communication. The last option, an additional helper thread, rarely improves performance for small messages owing to excessive context switching. For communication-heavy applications with frequent large messages it has been shown to pay off [39]. However, naturally this involves sacrificing a valuable thread that should be used to perform computation. With enclave tasking, sufficient and well-timed calls to `MPI_Test` and `MPI_Iprobe` can be elegantly built in the algorithm and I detail three different approaches for doing so. I call these techniques *exchangers*.

First, I present the symmetric exchanger. This approach exploits the symmetry of the data transfer in ExaHyPE to post the `Irecv`'s as early as possible. When a cell issues an `Isend` to a neighbouring rank, it knows that it will receive one in return and posts the `Irecv`. This is because the Riemann solves are executed redundantly on neighbouring ranks. This exchanger avoids the late receiver pattern and is designed for an implementation using the rendezvous communication protocol.

The second approach is called an immediate exchanger. The `Isend`'s are issued immediately by the producer processing the boundary cells. Only, when an `Iprobe` detects that a message is available is a matching `Irecv` posted.

Finally the third approach that is widely used in HPC and therefore acts as the baseline is the aggregate exchanger. During the traversal, all the messages are pushed into a single buffer that is sent out en masse at the end. Once the receiver knows the length of the whole buffer it posts an `Irecv`.

All three exchangers ensure message progression by plugging into the wait barrier at the end of the traversal, at which time the remaining STPs are processed. Other solutions issue a blocking wait at this point for pending communication requests with `MPI_Wait/Waitall`. Although such an approach is simple and ensures guaranteed progress, it wastes the computational resources of the master thread. Enclave tasking instead models a *logically blocking wait*. The main thread should first ensure that all available job consumers are running. Then in a loop it should progress the existing requests using `MPI_Test` and `MPI_Iprobe`, but additionally also process some tasks from the shared queue. This offers a distinct advantage over other applications, which either waste the thread with a blocking `MPI_Wait` or sacrifice a thread to constantly poll the MPI implementation.

Table 2.1: The benchmark's characteristics on one Broadwell core for a $27 \times 27 \times 27$ grid. Two values for arithmetic intensity (AI) as flops per byte are reported: flops per byte relative to L1 cache access [78] and relative to main memory access (DRAM) [41].

Order	Mflop/s	Bandwidth (MByte/s)	AI vs. L1	AI vs. DRAM
3	1464.35	568.18	0.08	2.38
6	2893.11	388.90	0.10	6.73
9	3111.02	199.60	0.11	14.28

2.4 Results

In this section I introduce the benchmark application I use to assess the enclave tasking idea. It is evaluated in comparison to a previous parallel implementation that serves as the baseline. I then introduce the three systems and the results on each. Throughout I discuss the implications and insights. The results are largely the same as those found in the original work [13].

The application studied is a standard benchmark, which models compressible Euler equations solved with explicit timestepping [53]. The equations have five unknowns and the initial conditions are shown in Figure 1.1. These are chosen such that the application does not exhibit any shocks yet showcases the dynamic AMR features of the code. The refinement criterion is simply gradient-based. If the solution is smooth then the surrounding mesh is coarsened, and if the gradient of the five unknowns exceeds a threshold the cell is refined. Otherwise the grid construction remains the same as outline in Section 2.3.1. The grid depth is constrained by $\ell_{\min}$ and $\ell_{\max}$ using a tri-partitioning scheme. The mesh refinement depth $\Delta\ell$, is defined as $\ell_{\max} - \ell_{\min}$. Leaf nodes of the tree contain Lagrange polynomials with Gauss-Legendre points. In this set of experiments polynomial orders $p \in \{3, 6, 9\}$ are studied for a range of runtime properties. The higher the order, the higher the workload and arithmetic intensity per enclave job. Order 9 is still lower than many other applications in HPC, yet represents a lower bound of anticipated perfor-

Table 2.2: Degrees of freedom values for all experimental set ups for various grid sizes, order and levels of refinement performed on a single machine. Values for $81 \times 81 \times 81$ with AMR are omitted as they exceed the available memory resources for a single node.

	$27 \times 27 \times 27$			$81 \times 81 \times 81$		
$\Delta\ell$	$p = 3$	$p = 6$	$p = 9$	$p = 3$	$p = 6$	$p = 9$
0	1259712	6751269	19683000	34012224	182284263	531441000
1	1642432	8802409	25663000	—	—	—
2	3346368	17934441	52287000	—	—	—

mance [40]. The solution in each cell with five unknowns is spanned by $5 \cdot (p+1)^d$ doubles. I use the three dimensional setup ($d = 3$). From these set ups bring an application dominated by the compute-bound P tasks. In Table 2.1 the effect p has on the runtime characteristics is shown. The R tasks are computationally cheap yet load large amounts of data from the main memory. Therefore the arithmetic intensity remains around 0.1. Increasing the value of p however leads to a much higher intensity relative to the main memory access. This attributable to the use of small arrays that fit into the low level caches, a key selling point of DG.

The degrees of freedom (DoF) depends upon on the number of cells, the number of unknowns and the polynomial order p. The DoF counts for all the experiments performed on a single machine are given in Table 2.2.

The explicit time stepping is based upon the ADER-DG scheme outline in Section 2.3.1 and the literature provides further details [12, 21, 75]. To combat the non-linearity of the Euler equations, $p + 1$ Picard iterations are used. This ensures that the P tasks contain sufficient flops and we do not see any effects of load imbalance among cells. The plain Rusanov solver from [53] is used on the faces in the R tasks which are solved directly in the traversal. The outcome of the P tasks is fed into the Rusanov solver to follow the ADER-DG predictor corrector scheme. Cells are not stored redundantly along MPI sub-domain boundaries. However, the computation of the R tasks is done on both neighbouring ranks as they exchange

the input for the Riemann solve. The outcome is then calculated redundantly. The application uses fixed global timestepping so all tasks use the same timestep after the first solve.

All result data includes only the timestep time, which neglects many influential yet irrelevant factors for this thesis such as grid setup, IO and load balancing costs. Unless noted otherwise, the simulation is allowed to run for 20 timesteps. This is long enough to ensure accurate results yet no dynamic AMR is triggered in this time which greatly simplifies calculating the DoFs and average time data. It has no real effect on the performance of the algorithm as validated experimentally. Normalised time is reported as the time per degree of freedom update per Picard iteration. Speedup metrics are compared against an optimised serial version of the code without shared or distributed memory parallelisation support and their respective overheads. Both metrics allow for comparisons to be drawn against different experimental set ups such as grid size and polynomial order.

In all cases, the code must effectively manage the potentially millions of tasks produced, where each has completely different compute characteristics. While few real-world examples exist where the grid changes at every timestep, the choice of experiments represents a lower bound on the performance of enclave tasking. If some of the assumptions and constraints are loosened, severe performance engineering can be undertaken to quantitatively improve the results presented.

To evaluate the performance of enclave tasking, it is compared to the previous implementation based on conventional TBB parallel fors. In this implementation, enclaves are processed one after another in a series of parallel fors, and the boundaries processed serially [77]. The two methods make for a good comparison: enclave tasking substitutes a series of synchronisation points for a task management overhead.

Two different machines are used to perform the experiments. The first is a conventional multi-core cluster whereas the second features many-core processors.

These are anticipated to be the building blocks of exascale applications. The specifications are:

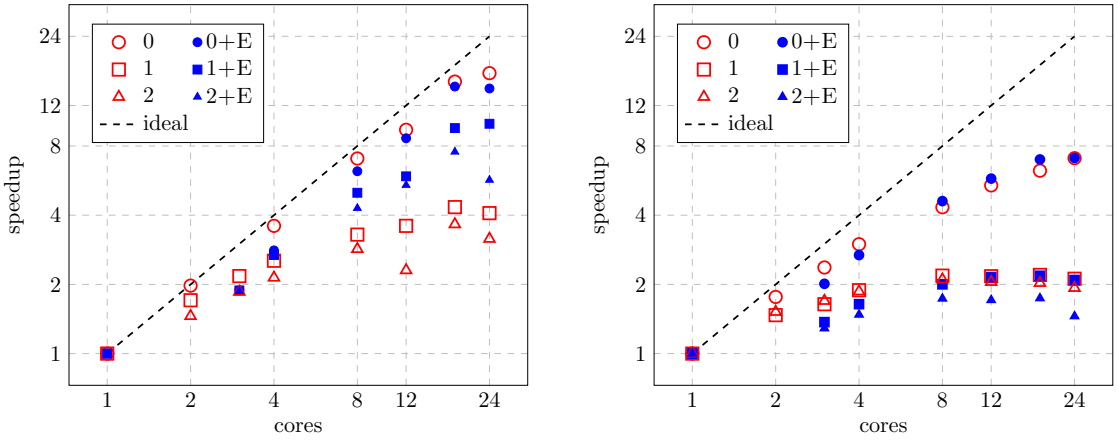
a) **Hamilton**. An Intel E5-2650 v4 (Broadwell) cluster with 2×12 cores per node. They run at 2.20 GHz up to 2.80 GHz TurboBoost and are connected by Omnipath. 64 GB TruDDR4 memory is available per node, with 256 kB L2 cache and 30 MB L3 cache. All counter measurements were reported by this machine with Likwid [74].

b) **CoolMUC3**. A Xeon Phi 7210-F (KNL) cluster with 64 cores per node and up to 256 Hyper-Threads, configured in the quadrant clustering mode. The cores run at 1.30 GHz up to 1.50 GHz with TurboBoost and are also connected by Omnipath. 96 GB DDR4 memory (80.8 GB/s) is available per node with 16 GB High Bandwidth Memory (460 GB/s) per node available as a cache. The on-chip cache configuration is as follows: on level 1 there are 64×32 KB 8-way set associative instruction caches and the same for the data caches. On level 2 there are 32×1 MB 16-way set associative shared caches.

For compiling the code I used the tools provided by Intel's Parallel Studio 2017. In particular their TBB library for shared memory support along with Intel MPI for distributed memory message passing. I now present the experimental results, in order of increasing machine scale. First I demonstrate good scaling on a conventional multicore machine. I then show that enclave tasking along with many other codes suffer from the more complex many-core machines. Finally, I compare the three exchangers outlined in Section 2.3.4 and then demonstrate the distributed-memory scaling up to 756 cores.

2.4.1 Multicore Shared Memory Scaling

The first results presented with enclave tasking (Figure 2.7) validate many of the hypotheses made. It is clear that given a regular grid, the enclave idea does not



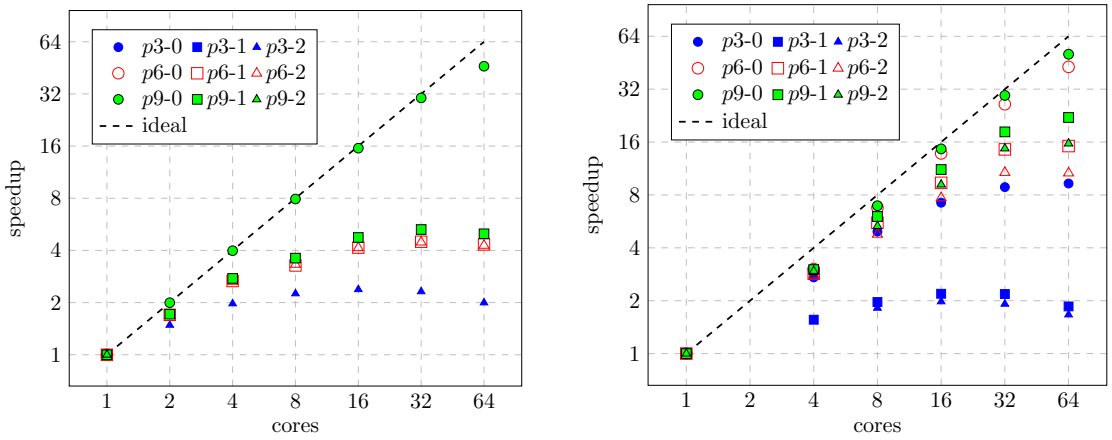
(a) $p = 9$ (b) $p = 3$

Figure 2.7: A best ($p = 9$) and worst ($p = 3$) case comparison of the shared memory speed up with (+E) and without enclave tasking on a multi-core machine (Hamilton). The numerical labels refer to the permitted number of refinement levels allowed. A value of zero denotes a regular grid.

justify its task management overhead. Instead, a single large parallel-for over all cells yields better results. However, as soon as adaptivity is brought in, enclave tasking shows its true potential. By exhibiting only a single synchronisation point per timestep it is able to scale to many more cores than the parallel-for based solution provided the order is sufficiently high (Figure 2.7a). It is able to keep all cores busy, while the producer processes the AMR skeleton cells. The extraction of regular sub-grids severely limits the concurrency of the non-enclave approach. With higher levels of non-localised adaptivity with $\Delta\ell = 2$, the ratio of skeleton cells to enclave cells drastically increased and reduces the potential concurrency. For a high enough order and therefore high computational load per P task, this has a reduced impact. The difference can be seen in Figure 2.7a compared to Figure 2.7b.

2.4.2 Manycore Shared Memory Scaling

The results from the previous experiment are reran this time on a the KNL many-core architecture with 64 cores per node. The story is largely the same, with enclave tasking proving effective at combating the dependency challenges of adaptivity (Fig-



(a) Parallel-for (b) Enclaves

Figure 2.8: A comparison of the two approaches on a many-core machine for a range of polynomial orders $p \in \{3, 6, 9\}$ and levels of adaptivity $\Delta\ell \in \{0, 1, 2\}$.

ure 2.8). Although the code benefits from two threads per 2-core tile, oversubscribing two threads onto one core did not pay off. Again, strong scaling effects can be seen even with low core counts unless the order is sufficiently high ($p = 9$).

Although the scaling looks smoother in Figure 2.8b, a closer look shows that only for a small subset of the experiments with fairly regular grids and high order does the code increase by a factor larger than 32. With clock speed over 50% less compared to the multi-core architecture in the previous experiment, the use of a many-core architecture does not pay off with either approach. While the many-core architecture is built for high intensity workloads such as the high order P tasks, the performance is polluted by the irregular data access of the R tasks and dynamic adaptivity. The high bandwidth MCDRAM used in cache mode does help somewhat with these issues, and contributes to the smoother scaling than that of a multicore architecture. Ultimately, with codes featuring high intensity workloads and regular data access few and far between, it is no surprise that since undertaking these experiments Intel has chosen to discontinue their many-core architecture [42].

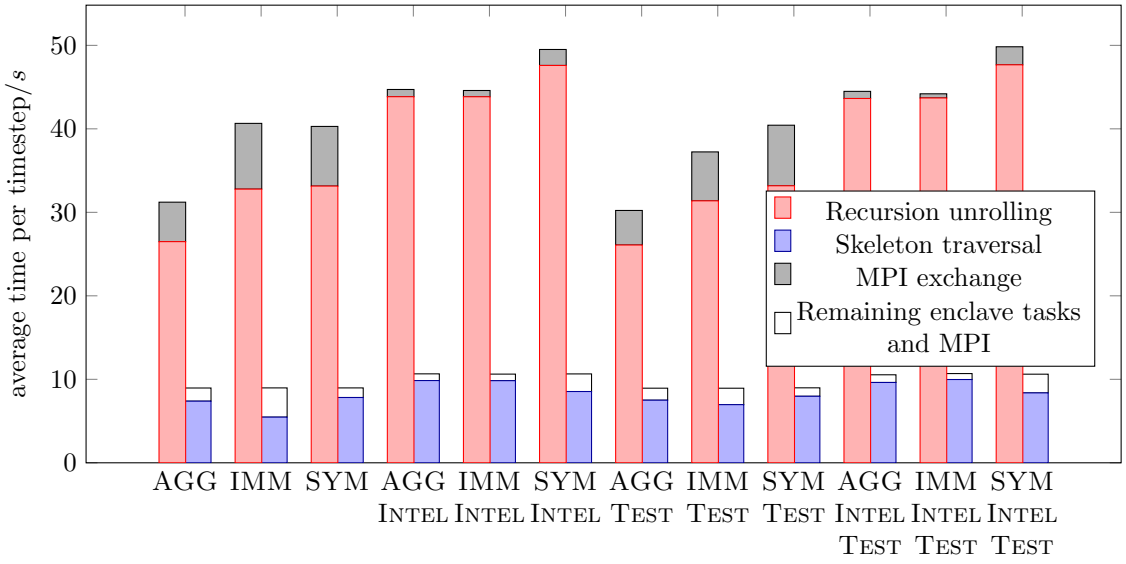


Figure 2.9: Comparison of baseline implementation (left stacked bar) with enclave partitioning (right stacked bar). Runtimes for the Euler benchmark with $p = 6$ on regular grid hosting 182,284,263 dofs on 30 ranks. AGG, IMM, SYM are three different send/receive strategies. Intel denotes the usage of Intel's MPI progression thread. Test denotes manual MPI progression through `Test` calls.

2.4.3 MPI Data Exchange Configuration

To efficiently scale up the code onto multiple machines, I now evaluate the three MPI data exchangers introduced in Section 2.3.4. In ExaHyPE and by extension most DG codes, the communication between neighbouring cells along sub-domain boundaries makes up for most of the data transferred over the network. Therefore the negligible effects of other communication such as time-step size reductions can be ignored here. The three exchangers are Symmetric (Sym), Immediate (Imm) and Aggregate (Agg). To summarise, Sym exploits symmetry in the grid to issue early matching receives for sends. Imm issues sends immediately during the traversal that are picked up by the receiver lazily using a probe. A matching receive is then issued. Finally, by following the conventional approach of pushing all data into one large buffer to send off at the end of the traversal, the Agg exchanger provides a methodological baseline [37].

The application is the same one used for the shared-memory experiments, al-

though the grid size is increased to $81 \times 81 \times 81$ to give 27^3 cells per MPI rank with 27 ranks. ExaHyPE introduces two management ranks which again can be ignored. The results in Figure 2.9 compare the exchangers for $p = 6$. I additionally compare enclave tasking against the parallel-for approach.

An excellent result for enclave tasks is that the communication is effectively hidden by the processing of the asynchronous tasks. Even better, the tasks are processed throughout the traversal. This is shown in Figure 2.9 by the small area of white bars. The findings from the shared-memory results are repeated here: the parallel-for approach takes much longer to complete the traversal as it must also compute the P tasks directly.

To further improve the performance of the data exchange, two optimisations are introduced denoted by Test and Intel in Figure 2.9. The Test optimisation additionally polls `MPI_Test` on pending `MPI_Request` objects throughout the traversal. In both methodologies `MPI_Test` and `MPI_Iprobe` are called repeatedly to progress outstanding messages. The Intel optimisation signifies the use of the Intel feature providing a helper thread to manage the asynchronous communications. It is enabled via the `I_MPI_ASYNC_PROGRESS` environment variable. In this experiment the ranks are pinned to a socket each (via `I_MPI_ASYNC_PROGRESS_PIN`), so each rank must sacrifice a core from the socket to the progress thread. This is one of the limitations that enclave tasking seeks to prevent.

The effect of maximising the communication window with enclave tasking means that the different exchangers and optimisations have little impact on the results. The communication wait time is fairly minimal already. However, the effects of the optimisations can be more clearly seen for the parallel-for based implementation. The Test option has limited impact, as the Intel MPI implementation seems capable of asynchronously transferring data with only minimal polling required to progress messages. The trade-offs presented by the Intel optimisation are clear. By sacrificing a core per rank to MPI management, the communication time may be reduced

yet the traversal time increases.

With Imm and Agg performing well in all cases, it seems the late receiver pattern with a probe to pick up the message followed by an `Irecv` has to be avoided for other reasons than runtime. Although the Imm exchanger is slightly more efficient than the other two in this experiment, there are other properties to consider. For larger numbers of ranks or applications with more complex PDEs, the memory requirements of Imm may go up dramatically. This is because of its reliance on a late-receiver pattern and therefore potentially extensive buffering on the sender or receiver side, with rendezvous or eager protocols used respectively. The Agg exchanger however adds extra memory buffers and transfer costs while providing little benefit for enclave tasks. The parallel-for approach benefits from the aggregation as the buffer is completed at the end of the traversal and the data is sent with minimal latency. The network architecture may also play a role in the choice of exchanger. This experiment was ran on the Omnipath-based Hamilton cluster, which is capable of handling the many small messages issued by Imm and Sym. When performing additional experiments (not shown) on an older Infiniband-based cluster, the Agg exchanger is the better choice. In ExaHyPE the default is to use enclave tasking in tandem with the Imm exchanger yet providing the option of the slightly less performant yet more memory efficient Sym exchanger for larger set ups. On older Infiniband systems the Agg exchanger should be used. These options are available at compile time.

Now that the optimal configurations for the MPI exchange has been found, I move onto the results from up-scaling the application onto multiple nodes of the Hamilton cluster.

2.4.4 Hybrid MPI+TBB

In this experiment, the grid size is increased to $81 \times 81 \times 81$ such that each rank gets the same number of cells as in the shared-memory experiments. The refinement

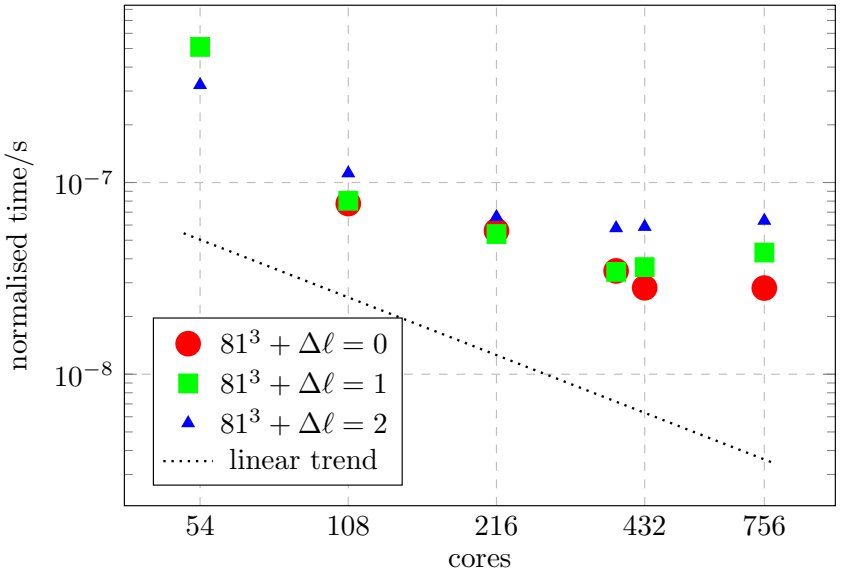


Figure 2.10: Normalised time (per dof per grid sweep) for $p = 6$ on the Ivy Bridge cluster. The trend line denotes a linear speedup. $\Delta\ell$ denotes the number of added adaptivity levels relative to a base grid of $81 \times 81 \times 81$.

criterion remains the same. The results with $p = 6$ show that the same linear trend of shared-memory scaling can be seen when distributing the domains across machines (Figure 2.10). Although the trend is offset from a real linear speedup, this is the expected cost of introducing the TBB and MPI routines. Future work will focus on lowering the overhead introduced by these.

An important feature of these results is that the scaling remains similar for increasing levels of mesh refinement. The performance is somewhat reduced attributable to sub-optimal distribution and load balancing of cells among MPI ranks. Codes with adaptivity is where enclave tasking is designed to shine. Although any good scaling code has to hide the communication behind computation, this is often achieved manually by processing those cells along sub-domains first, sending out the data and then continuing with internal work [75]. However, if a cell along the boundary decides to trigger a refinement, the sending out of all data will be held back by the refinement procedure. In enclave tasking, while the producer sends out data one by one and may refine cells dynamically, the remaining cores are kept busy by the jobs in the background queue. The only caveat to this is that enough inte-

rior cells must have already been processed and therefore spawned their jobs to the queue before any delays are incurred by complex skeleton processing. If the ratio of enclave to skeleton cells is high, this drawback will not materialise owing to the underlying traversal following a Peano curve [77].

2.5 Shortcomings Of The Presented Approach

The shortcomings of the approach come from the design decisions made to suit the intended context. Although the adaptivity may be fully dynamic, the approach performs best when the adaptivity is localised in the grid such that large enclaves may form. If the ratio of skeleton cells to enclave cells is too high then there will be insufficient work to be executed by the other threads in the background. This materialises when the shared work queue empties before the end of the traversal. Applications can tailor their refinement to avoid such random non-local patterns.

As mentioned previously, the approach does not pay off if the grid and therefore implicitly created task graph does not change. Applications not requiring a dynamic grid should create the task graph once. The result can be passed to a scheduler for an optimal ordering to be used for the entire simulation. Experiments in Section 2.4 show that a standard parallel for over a grid without any refinement is more efficient.

Another design decision was to embed the R tasks into the grid traversal to ensure no bursts of bandwidth-bound operations occur. Although this provides a clear advantage, the approach only works if the timestep size is known or can be estimated. Since the P tasks determine the timestep size, a sophisticated time-stepping scheme with rollbacks must be used [12]. Such rollbacks become mandatory for non-linear PDEs, which may unpredictably trigger refinement invalidating an enclave.

The final shortcoming of this work is that distributed memory load balancing and tailored task placement were considered out of scope but an area of future work.

Some ideas to tackle these challenges are presented in the outlook and final chapter of this thesis. Although both techniques would change the results quantitatively, the outcomes remain qualitatively valid.

2.6 Outlook

Enclave tasking has proven to be an excellent fit into the design of ExaHyPE, a ADER-DG code with matrix-free explicit timestepping. By spawning tasks during the mesh traversal, high levels of concurrency can be ensured while performing dynamic adaptive mesh refinement and MPI communication. No task graph has to be set up. The approach tackles the previous implementations of limited scalability on adaptive meshes and efficiently scales on multi-core, many-core and distributed memory machines. By mixing compute and bandwidth bound tasks, enclave tasking relieves pressure from the system's critical resources.

An area for future research is deployed the regions of enclave tasks to an accelerator. Sundar and Ghattas coined the term enclave tasking [71], ensuring that enclaves on different accelerators do not have to communicate. It is well suited to such hardware, as the data transfer cost may yet again be hidden behind the computation. This is one of the key selling points of asynchronous computation in HPC. On shared-memory systems, enclave tasking shows the benefit of fusing multiple tasks (or jobs) together into batches. This would be a useful feature to be supported natively in the task runtime. A comprehensive priority system for tasks should also allow for more efficient scheduling of work between skeleton and enclave cells.

On distributed memory systems, I compared three approaches to asynchronous data exchange and show the potential of enclave tasking as a methodology overlapping communication with computation. Although the immediate exchanger, firing off data in small packets, proved most efficient on Omnipath systems, a callback

mechanism for handling unexpected messages on the receiver side would greatly reduce the memory buffering requirements. Modelling the exchanger using the one-sided features of MPI-3 may prove effective in this scenario.

Overall, while the results showcased impressive scaling accounting for the complex requirements of the ExaHyPE codebase, the implementation still requires performance engineering to reduce the overheads imposed by the task and message runtimes. A native implementation based on plain threads would eliminate the need for the complex TBB task scheduler. Increasing the ratio of enclave to skeleton tasks would reduce the chance of thread starvation and increase the throughput of tasks. Such an approach would have to more aggressively identify areas of the grid to form enclaves. With ExaHyPE this could be done by tightening the constraints upon the dynamic refinement capabilities of the code. For example if it is known that dynamic refinement will not happen for a cell along a resolution boundary then the cell may be included as part of an enclave. Likewise, the user applications could utilise more intelligent refinement patterns that excessively refine to maximise the size of enclaves and minimise grid refinement phases of the simulation. Such an approach would be a trade off between increased cell counts over improved throughput and scaling. An efficient load balancing of the tasks shared among cores and machines is also an open topic. An idea for diffusive load balancing based on the asynchronous teams introduced in the following chapter is discussed in Chapter 4.

Chapter 3

Asynchronous Teams

In many areas of large-scale computing, the success of a given approach is preliminarily dictated by the performance. At exascale, a second success factor enters the game: fault tolerance. Although quantifying the performance of a given solution is usually trivial, proving a tolerance to "faults" is challenging. Faults in this thesis refer to errors in hardware, such as outright failures or "silent" memory faults that produce erroneous results. Barring HPC, almost every other field uses the idea of replicated resources to ensure that maximum performance is achieved even when faults occur [10]. For example, cloud computing, sensor networks, desktop grids and peer-to-peer networks all base their fundamental fault tolerance capabilities upon the idea of replicating resources. Traditionally HPC has relied on a technique called *checkpoint-restart*. It requires applications to periodically save their state to an external storage medium that can be loaded in the case of a failure. Faults then reduce the efficiency of an application, forcing rollbacks previously saved states. The success of this approach is owing to three assumptions that have remained valid over the past 40 years [28].

1. Application state can be saved and restored much quicker than a systems mean time to interrupt (MTTI).

2. The hardware and upkeep (e.g. power) costs of supporting frequent check-

pointing are a modest (perhaps 10–20%) compared to the systems overall cost.

3. System faults that crash (fail-stop) the system are rare.

Work over the past decades in checkpoint recovery has largely focused on keeping these statements valid. However, each one is challenged by exascale. Although the mean time before failure of a single node is measured in years, combining the large amount of nodes required to form an exascale machine reduces this value dramatically. This is a troubling issue when the purpose of such machines is to allow for much larger experiments to be carried out, which naturally increases the time required to write a checkpoint. Several theoretical studies have been carried out in the past decade that validate with current failure rates, replication will be mandatory for an exascale application [9, 24, 25, 28, 59]. Without it, exascale machines will spend most of the time writing and recovering from checkpoints rather than computing "useful" results. This is achieved by modifying existing scaling models such as Amdahl's [1] and Gustafason's [36] Laws to include reliability as outlined by Zheng and Zhiling [80].

In the exascale software roadmap by Dongarra et al [18], replication is again raised as a potential solution to the fault-tolerance challenges at exascale. Following up on these ideas, several empirical studies presented implementations of process replication and a thorough review is undertaken in Section 3.1.

In addition to (unpredicted) increasing hardware reliability, the issue process replication faces to this day is that multi-million Dollar investments into exascale computing is a difficult sell for vendors when requiring 2–3x redundant computation. Replication based fault tolerance has since fallen out of fashion, replaced instead by approaches that catch or mitigate faults. Run-through-stabilisation techniques pioneered by the ULFM-MPI project [7, 15, 27] allow applications to use new MPI features to continue running even when a process fails. Other alternate ideas include process migration teamed with failure prediction. Faults are predicted using

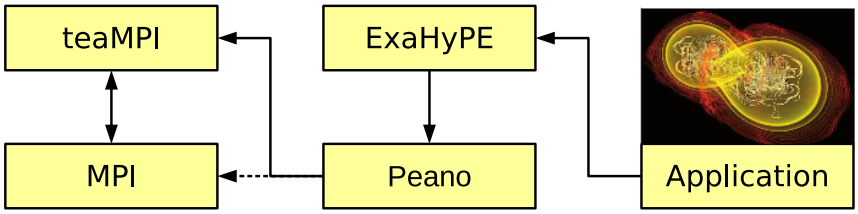


Figure 3.1: The layers of a proposed software stack involving the teaMPI library. Both ExaHyPE and Peano are optional. If the teaMPI layer is removed the application will function identically to when replicated.

advanced detection techniques and processes are migrated before it occurs. Although a large and promising research field, it is still not capable of handling all errors and therefore still falls back onto traditional methods [11, 31–33, 67, 68, 76, 81].

The replication research proposed here will tackle, in parallel, another major show-stopper raised in the exascale roadmap: heterogeneous system performance. Even individual nodes may differ by up to 10% through effects such as component binning and heat. Therefore, one of the main contributions of this project will be how this imbalance can be reduced with replication. Clearly, for process replication to be adopted it must also provide resilience without performance deficits.

In this chapter I present a new library, teaMPI, in which **tea**ms of **MPI** processes are created transparently from the application to provide fault tolerance and detect performance issues of team members [38]. I first define some of the key terms that will be used throughout the chapter. Since the original idea of replicating MPI processes is not new I carry out a review of the existing implementations in Section 3.1. After a critical analysis of these approaches, I introduce my library and its novel approach surrounding teams *without synchronisation* in Section 3.2. This has several advantages over existing approaches such as the ability to detect faults and performance issues of individual ranks with minimal overhead, as discussed in Section 3.3. The chapter ends with some concluding remarks.

teaMPI overview

Rank replication is enabled by splitting a global pool of ranks into distinct sub-pools called teams. Each team operates as a separate instance of the application by exchanging messages through a new communicator. At application-defined points, named heartbeats, the teams asynchronously exchange information to compare the states of data and performance.

3.1 Review Of Existing Approaches

A review in the area of existing replication work investigates the challenges faced by previous implementations and the solutions each one presents. As the number of such projects is large and vary with intended use, the focus is on supporting process replication in an MPI environment used by most scientific computing codes. First, I explain the core features common to all approaches. As each share a common goal they naturally share features. However, the approaches differ in each project with varying degrees of success. In Section 3.1.3, I next introduce the projects and which features they pioneered or innovated. Finally, the strengths and weaknesses are critically evaluated to direct the contributions of this project.

3.1.1 Integration Of Replication

In all the existing work I found, the underlying replication aims to be as transparent as possible to the user's application. This is common to many other areas of replication, such as the RAID data storage virtualization technology. When saving files redundantly it is not necessary for the user to know that these multiple copies exist.

To realise this, MPI replication is implemented as a library that intercepts MPI calls to handle the replication internally. The requirement to intercept MPI calls is not unique to this area, therefore the MPI standard provides a dedicated "profiling"

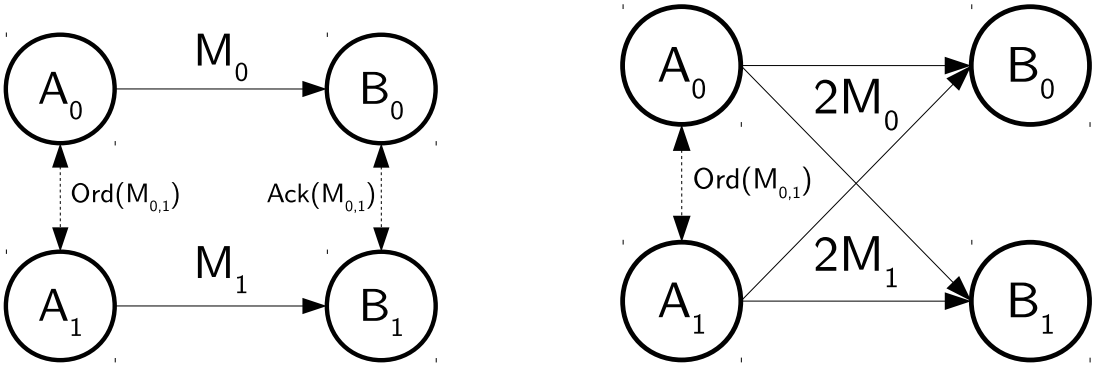
interface: PMPI. The profiling layer of MPI is designed such that libraries can intercept the calls to the MPI library and execute arbitrary code instead. Usually the interceptor forwards the MPI invocation to preserve the function's semantics. The MPI standard requires that the core functionality be provided by "name shift" functions prefixed by an upper-case P. Each function is declared twice. For example, `MPI_Send` is also declared as `PMPI_Send`.

MPI library implementers have two approaches given by the standard to provide this interface. If the compiler and linker support weak symbols (as most modern systems do), then the original MPI functions can be weakly defined. At link time the symbols are overridden if another library provides a "strong" definition for them. When an application calls `MPI_Send` (or any other function declared in the libraries' header) the definition is provided by the library. The original functionality of `MPI_Send` is then available via the `PMPI_Send` interface.

The other solution relies on the C macro preprocessor, and therefore the library cannot be added at link time. I do not outline this usage here as it is not required for modern systems. Interested readers are directed towards Section 14.2 (Profiling Interface) of the MPI Standard [55].

For replication, this allows libraries to operate completely transparently. If an application requests the number of ranks available to it through `MPI_Comm_size`, the library can simply return the value given by `PMPI_Comm_size` and divide by the level of replication.

One of the main disadvantages of the profiling layer is that it can only be used by a single library at a time. Only one may intercept the MPI calls by overriding the weak symbols. A workaround is provided by the P^NMPI project [69, 70], which allows multiple applications to use the profiling layer concurrently. Another potential downfall for library developers is that with hundreds of MPI functions, missing out any will result in application bugs. Thankfully, it is possible to generate the interface for a library automatically using the wrap script developed by Gamblin [34].



(a) Parallel (b) Mirror

Figure 3.2: A comparison of the parallel and mirror consistency schemes. There a two ranks, A and B, with a replication factor of two. Rank A sends a message to rank B. The mirror mode must ensure that the ordering of messages is identical for wildcard receives. Parallel mode must additionally acknowledge that the messages were received.

3.1.2 MPI Consistency

For any area of replication, keeping the state of replicas *consistent* is an important topic. Specifically for MPI, the state refers to the result of calculations and the data sent between ranks. It must be identical among replicas. Existing approaches adopt ideas from the wider field of fault tolerance, using active or passive replication. Active replication is the scheme used by most, where messages are sent to all replicas. This is in contrast to passive replication where a master replica is chosen that then broadcasts results to slave replicas.

Two main protocols exist to ensure replicas remain consistent: mirror and parallel (Figure 3.2).

Definition 3.1: Mirror Consistency Protocol

Each message is duplicated to all replicas redundantly. For each message r^2 are sent in total, where r is replication factor.

Definition 3.2: Parallel Consistency Protocol

Each message is only sent to the corresponding replica. For each message r are sent in total, where r is replication factor.

The mirror protocol requires ranks to send/receive from all of its replicas, whereas in the parallel mode ranks only send/receive with their respective replica pair. Mirror sends $m \cdot r^2 + c$ messages, where m is the number of original messages, r is the replication factor and c is the small messages sent to ensure consistency among replicas. The parallel protocol is more bandwidth efficient, sending only $m \cdot r + c$ messages. However, the parallel protocol has to send many more messages than the mirror protocol to ensure consistency. This leads to a larger value of c.

The mirror protocol only has to send extra small messages to ensure that MPI behaves identically across replicas. Certain features of the MPI standard are allowed to behave non-deterministically. Non-deterministic message passing means that a message may not be sent the same way in two different runs of the application. This subset of MPI functions raise issues with ensuring consistency among replicas, as executing the same code on each replica may return different results. The first example of this is the `MPI_ANY_SOURCE` parameter in receive operations, that could allow replicas to receive messages in differing orders. Other features which exhibit similar properties include: `MPI_Wtime`, `MPI_Probe`, and `MPI_Test`. The replicas communicate to ensure that the operation occurs identically.

The parallel consistency mode must also ensure that messages are sent. This requires introducing many small messages into the system testing for completed operations, but reduces the strain on the network bandwidth.

Micro-benchmarks that are either bandwidth sensitive or message rate sensitive stress both the mirror and parallel consistency schemes respectively. The mirror protocol suffers from large message sizes due to bandwidth constraints whereas the parallel protocol suffers from small message sizes sent at high rates where latency becomes a bottleneck [29]. When tested with multiple real world applications, it was clear that most exhibit properties of the former, with larger messages sent less often. Therefore the overhead of the mirror protocol is much larger than the parallel protocol for many real world applications.

Passive replication is less popular for MPI, used by only a single existing implementation. This is because the result of operations must be broadcast to the cloned slaves. Although it allows a library to easily vary the number of replicas per rank, the increased latency is too greater price to pay.

3.1.3 Previous Implementations

In total I found five existing implementations matching my criteria, each taking a different approach to provide replication. Two of these projects, rMPI and MR-MPI, merged to form redMPI. None of the existing approaches are under active development, or to the best of my knowledge, still supported.

rMPI

The rMPI project was the first empirical study on replicated MPI processes and operates using the PMPI [29]. In theory this should allow rMPI to work regardless of MPI version. However, the prototype includes a number of MPICH specific function calls to implement collective operations in a point to point manner and as a result is tied to a fixed version of MPICH. This is likely a result of being first out of the gate, as a number of competing implementations were also released following rMPI in 2011. After reiterating the need for examining replication as a solution for resilience, the authors dive into the practical details of replicating MPI process by introducing the two supported active consistency modes: mirror and parallel.

The library differentiates between master and replica messages by flipping the higher order bit of the message tag. The implementation is therefore limited to only duplex replication. It is also unable to support the simultaneous use of the `MPI_ANY_TAG` and `MPI_ANY_SOURCE` parameters. Finally, it relies on the (commonly valid) assumption that the underlying MPI implementation supports tags larger than the 2^{15} required by the standard.

MR-MPI

The modular redundancy MPI project (MR-MPI) from Oak Ridge National Laboratory [26] can be seen as a polish of the rMPI prototype, freeing itself from a number of its predecessors limitations. Although MR-MPI only supports the mirror protocol, it is MPI implementation independent and supports arbitrary replication, both fixed and partial. With partial redundancy, some ranks may have more replicas than others. MR-MPI also natively supports collective operations by using the `MPI_Reduce_local` function. Ultimately, performance results from their work largely mirror the findings from the rMPI project.

redMPI

The redundant MPI (redMPI) project is a collaboration merging the two previous libraries: rMPI and MR-MPI [8, 23].

Although work on rMPI and MR-MPI carried out performance testing and provided some initial results, there were many unanswered questions from the theoretical studies to tackle. First, the authors define a mathematical model to demonstrate the effectiveness of replication combined with checkpoint-restart over the pure checkpointing solution [23]. MR-MPI did include partial redundancy, so redMPI did too. However, after an investigation it was found to not be an effective solution for fault tolerance. The second focussed on error detection and correction [30]. In this paper the authors define a new approach where replicas only send an envelope containing a hash of the message content rather than the actual message. If a replication factor of two is used, the error can be detected if the two hashes are different. A replication factor of three allows for error correction as the correct result can be decided by a simple voting mechanism. This assumes errors are rare and therefore only one replica will be at fault.

EchoMPI

The EchoMPI project [16] was also released in 2011, but adopted a consistency scheme from fault tolerance traditionally known as passive replication in other fields. It is another library that proves the versatility of the PMPI, providing a library that is independent of the MPI implementation used. Passive replication implies that the state from one replica is transferred to all others. The library introduces the concept of "master ranks", which broadcast the result of MPI operations to "clone ranks". This allows for arbitrary yet simple replication of individual ranks, but introduces much larger latency overheads. Although a message only has to be sent once among master ranks, it must then be broadcast to the clones. Therefore limited bandwidth is saved at the expense of much high latency. The clones still do all computation, as the necessary data is received from the respective masters. To ensure complete coverage of the MPI library, the EchoMPI interface is generated using wrap [34]. The authors present future use cases for this style of replication such as parallelising heavyweight profiling tools, investigated in a follow-up paper [61].

Send Determinism Replication (SDR-MPI)

The SDR-MPI project developed at INRIA differs to those discussed previously in that it trades transparency and portability for simplicity and performance [51]. SDR-MPI modifies the MPI implementation, OpenMPI, to provide the consistency among replicas. It then enforces an ordering of messages at a lower level which requires marking the start and end of each communication between ranks. Although it increases the efficiency of the mirror protocol, the requirement of a patched MPI library is not worth the marginal gains.

The authors then later extend this implementation to allow replicas to share tasks among themselves with custom constructs [60]. Applications create tasks through an API and define sections in which they are executed. Within the section, tasks are shared among replicas such that computation is not performed redundantly. At

Table 3.1: Feature comparison of existing rank replication approaches.

	rMPI	MR-MPI	RedMPI	echoMPI	SDR-MPI
Fixed replication	✓	✓	✓	✓	✓
Partial replication		✓	✓	✓	
PMPI	✓	✓	✓	✓	
Error correction			✓	✓	
Work-sharing					✓
Overhead (1–5)	3/4	3/4	3/4	5	2

the end of each section the replicas coordinate their results.

Three issues exist with the SDR-MPI approach. First, as soon as the replicas no longer carry out computation redundancy then all benefits of replication-based fault tolerance are eliminated. Second, the work is split statically and does not take into account heterogeneity among tasks. Finally, it requires applications to be rewritten in a task language using their API and a custom MPI library. This imposes a lot on the application developers.

3.1.4 The Takeaways From Existing Approaches

A summary of the features for each library is presented in Table 3.1. The first project to benefit from existing work was the redMPI project, which analysed the strengths and weaknesses of its predecessors (rMPI and MR-MPI), forming a much improved library with a superset of useful features.

The first major learning point from the literature is that a "mirror mode" approach is problematic in a high performance environment because of the extra bandwidth overhead it incurs. Even though the "parallel mode" is a much smarter approach, the existing implementations add too much latency overhead by excessively communicating among replicas. In the following section I introduce a novel idea to drastically reduce the impact of this.

The second takeaway, specifically from the redMPI project, is that partial replication is not likely to be a useful feature as the authors discuss in [23]. As the

location of errors is usually unpredictable, it neither helps with resilience nor performance and only adds complexity. Therefore we do not build this feature into the library.

From the echoMPI library it is clear to see that an enforced master-clone topology with traditional passive replication is also not a suitable candidate in a high performance environment. The requirement to broadcast every result to clones and have an MPI communicator per rank is much too heavyweight. It is conservative in bandwidth overhead but adds large latencies per message. Although the benefits of this approach is that fine-grained replication can be achieved, as previously noted this is unlikely to be required. Clearly, the passive replication approach from classical fault tolerance is not a good fit for high performance applications. Instead an active scheme should be adopted where the replicas are responsible for their consistency, effectively resembling the parallel replication protocol.

The SDR-MPI project shows the potential of work-sharing among replicas, but the extensive changes to user code and the underlying MPI library nullify any potential use cases. In this work, I show that it is possible to get the same increase in performance with a PMPI wrapper that requires much less modification of user code.

3.2 The teaMPI Library

In this section I discuss the architecture and some key design decisions surrounding the teaMPI library. The implementation builds upon the lessons learned from studying the previous projects in Section 3.1: the replication should be as lightweight and transparent as possible. To deliver the replication, a C++ library is implemented in the MPI profiling layer (PMPI) similar to the existing implementations in the literature and is detailed in Section 3.1.1. In contrast, the SDR-MPI project requires applications to change large amounts of code to enable replication. Using the

PMPI layer provides two key advantages: (i) there is no requirement to modify the existing application code and (ii) allows the library to operate independently from the underlying MPI library implementation.

Algorithm 3.1 Splitting the `MPI_COMM_WORLD` communicator into teams.

```
1  // Duplicate MPI_COMM_WORLD first following recommended practise
2  PMPI_Comm_dup(MPI_COMM_WORLD, &TMPI_COMM_DUP);
3  PMPI_Comm_size(TMPI_COMM_DUP, &world_size);
4  PMPI_Comm_rank(TMPI_COMM_DUP, &world_rank);
5  team_size = world_size / number_of_teams;
6  // Calculate which team this rank belongs to
7  int team = world_rank / team_size;
8  // Split TMPI_COMM_DUP into number_of_teams sub-communicators
9  PMPI_Comm_split(TMPI_COMM_DUP, team, world_rank, &TMPI_COMM_TEAM);
10 // New rank and size returned to the application when called
11 PMPI_Comm_rank(TMPI_COMM_TEAM, &team_rank);
12 PMPI_Comm_size(TMPI_COMM_TEAM, &team_size);
```

The main idea of building the teams is to split a pool of ranks into teams of replicated ranks. The application has no knowledge that these replicas exist. The ranks are mapped into t equal sized teams of contiguous processes, lines 7–9 of Algorithm 3.1. This default mapping can easily be modified to any one-to-many function. To separate and de-synchronise replicas, the original MPI communicator `MPI_COMM_WORLD` is first duplicated for preservation, as recommended by the MPI standard for libraries, then split into t sub-communicators. Each team then has access to a separate communicator, `TMPI_COMM_TEAM`, for messages that would usually get sent via `MPI_COMM_WORLD`. This means that identical messages sent by different teams are not subject to the strict message ordering imposed by MPI, where messages of the same tag and communicator cannot overtake. The code snippet for splitting the communicator into teams is given in Algorithm 3.1.

An advantage of splitting the original communicator is that the mapping between real and logical rank identifier is handled automatically the MPI library.

Algorithm 3.2 The `MPI_Recv` function in the teaMPI library.

```
1  std::map<MPI_Comm, MPI_Comm> commMap;
2  int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
3               int source, int tag, MPI_Comm comm, MPI_Status *status)
4  {
5    // Map communicator, e.g. MPI_COMM_WORLD -> TMPI_COMM_TEAM
6    comm = commMap[comm];
7    // Perform receive with correct replica of source rank
8    // No synchronising consistency checks here!
9    return PMPI_Recv(buf, count, datatype, source, tag, comm, status);
10 }
```

As a consequence, this greatly reduces code complexity as many of the original MPI functions do not require large modifications. For all communication based functions the only change required is to map the provided `comm` parameter to one split for use in teams. If `MPI_COMM_WORLD` is used, then it is a simple mapping to `TMPI_COMM_TEAM`. However, if the application makes use of other communicators by splitting `MPI_COMM_WORLD` then `TMPI_COMM_TEAM` must also be split in the same manner. When an application splits the `MPI_COMM_WORLD` communicator, the library splits the `TMPI_COMM_TEAM` communicator and stores the updated mapping in a `std::map<MPI_Comm, MPI_Comm>`. The `MPI_Recv` wrapper function is given in Algorithm 3.2, where the only change is the communicator mapping. This also retains the correct values in the `MPI_Status` object. For example, the `status→MPI_SOURCE` field will automatically return the team rank of the source, not the global one. Other implementations such as rMPI instead are forced to make large changes for many functions. In rMPI each collective operation is replaced with a reduced efficiency point-to-point counterpart. The `MPI_SOURCE` field of `MPI_Status` objects must also be manually altered. Comparisons to other approaches cannot be made as each one does not publish their lower-level implementation details or make their code open source.

3.2.1 The Heartbeat Consistency Model

One of the key differences with the teaMPI library compared to the approaches detailed in Section 3.1 is in the way it ensures consistency among replicas. In all the existing libraries I investigated, each one enforces a *strong* consistency model (Definition 3.3).

Definition 3.3: Strong Consistency

The state of replicas is identical at every MPI function call.

This causes multiple issues, such as making sure that replicated non-blocking messages complete identically and that messages are received with the same source/-tag where wildcards are used. The paper introducing rMPI is an excellent collection of the issues faced by this model [29], also discussed in Section 3.1.3. Additionally, they implement the mirror mode as it more naturally supports fault tolerance (Definition 3.1, Figure 3.2b). However, the quadratic relationship between the messages and replication factor means that the bandwidth required is too high for the majority of codes. Network bandwidth is often a critical resource for HPC applications. Therefore I implement the parallel scheme, which only increases the bandwidth directly proportional to the number of teams. The rMPI project reported that this introduced many small messages into the MPI subsystem to ensure the other replica had sent its message. For `MPI_ANY_*` wildcards it also induces excessive synchronisation as the replicas must ensure each receive from the same source/tag.

Weakening The Consistency Model

Definition 3.4: Weak Consistency

The state of replicas is identical only at application defined points (heartbeats).

One of the core contributions of this work is that I weaken the consistency model (Definition 3.4). Most applications do not need to ensure a consistent state

among replicas on every MPI function call. Provided that the code uses MPI in a deterministic way, then linking against the teaMPI library will not lead to a change in the runtime behaviour. If the code uses non deterministic features such as `MPI_ANY_SOURCE`, then this statement may not hold. For example, if a receive is posted with a `MPI_ANY_SOURCE` source parameter the following may happen: one replica receives a message from rank x while another replica receives a message from rank y first. However, I assume that by the end of the iterations this will have no effect on the *overall* application consistency. The outcome is eventually deterministic.

Definition 3.5: Heartbeat

An asynchronous operation carrying data to compare the states of replicas.

Teams operate fully asynchronously, such that one may advance much further through the execution than others. Applications are only required to check-in via a "heartbeat" every so often, usually per iteration (Definition 3.5). As a "heartbeat" operation is performed fully asynchronously, it does not hold faster teams back. To keep integration cost of the library as minimal as possible for the user, this heartbeat can be inserted with a single line of code and still allows the code to compile and run as normal without linking to the teaMPI library.

Heartbeat Implementation

The heartbeat is realised via "hijacking" the `MPI_Sendrecv` function call such that with when passed `MPI_COMM_SELF` as a communicator parameter it operates as the heartbeat (Figure 3.3). teaMPI can differentiate between heartbeats and calls with the original semantics. Since a `MPI_Sendrecv` with the host rank (i.e. using `MPI_COMM_SELF`) is nonsensical, this is a safe choice. If for some reason an application does require this functionality, `MPI_COMM_WORLD` can safely be used. This function accepts many arguments with some perfect to leverage for teaMPI. The implementation of this heartbeat will be covered in the remainder of this section. However,

Algorithm 3.3 The `MPI_Sendrecv` function in teaMPI.

```
int MPI_Sendrecv(const void *sendbuf, int sendcount,
MPI_Datatype sendtype, int dest, int sendtag,
void *recvbuf, int recvcount, MPI_Datatype recvtype,
int source, int recvtag, MPI_Comm comm, MPI_Status *status)
{
  int err = 0;
  if (comm == MPI_COMM_SELF) {
    if (sendcount == 0) { // Consistency buffer not provided
      err |= heartbeat(sendtag);
      } else { // Consistency buffer provided
      err |= heartbeat(sendtag, sendbuf, sendcount, sendtype);
    }
    } else {
    // Perform actual SendRecv (unmodified parameters omitted)
    err |= PMPI_Sendrecv(..., mapComm(comm),status);
    mapStatus(status);
  }
  return err;
}
```

for now it is enough to know that the heartbeat is fully asynchronous (a regular `MPI_Sendrecv` operation is still blocking) and consumes very little bandwidth. First, I outline two initial uses of the heartbeat consistency scheme, the detection of (i) slow or (ii) faulty ranks. Then I show that the overhead of such a consistency scheme is minimal in most cases.

Measuring Performance Homogeneity

Algorithm 3.4 Compare progress of replicas

```
1: procedure comparePROGRESS
2:     for replica of this rank do
3:         Isend current time to replica
4:         Post Irecv for time from replica
5:     MPI_Testsome on pending requests check and progress received times
6:     Process times which have been received
```

The first intended use of the heartbeat consistency scheme is to detect slow ranks within a team. In Section 3.3, I show that this can pollute the performance of applications. Such slow ranks can be considered close to broken where a slow-down is a precursor to a failure. In general this means teaMPI must detect the performance variations among replicas. By comparing the heartbeat times among replicas it is then possible to detect if one is falling behind the others. This could be due to any number of factors such as variations in processor binning, heat levels or network traffic. Other replication libraries have no simple way to measure this because of the strict consistency protocols, where the states of replicas are synchronised at each MPI function call. Therefore it would have to be done on a much finer level at each function call which would introduce many small messages into the network sub-system. For example, measure which team reaches an `MPI_Wait` call first. This approach however will struggle to detect long running issues in ranks and will be sensitive to small variations in the individual runtime behaviour of ranks such as operating system interference. It also introduces extremely tight coupling between teams.

With teaMPI, the heartbeats are piggy-backed to carry timestamps. As mentioned previously, most scientific codes are iteration or loop based and therefore inserting the regular `MPI_Sendrecv` commands is just a single line addition in the loop body. Algorithm 3.4 outlines the steps required to compare the progress of replicas asynchronously using MPI non-blocking communication. It is imperative that synchronisation is avoided among ranks, in this case specifically replicas, so the library supports two modes of asynchronous communication. The first is used when the underlying applications supports the eager send protocol. Eager sends are where the data to be sent is transferred to a temporary buffer on the receiver until the matching receive is posted. It is then copied into the supplied buffer. Owing to memory and copy overhead on the receiver side, eager sends are only supported for small messages. The default threshold for the Intel MPI library is 256kB, for which the heartbeat messages easily fall under. If the receive is posted before the send then the `MPI_Testsome` operations ensure that the previously posted receives get completed. If eager sends are not supported by the MPI library or disabled by the user for their application, then these `MPI_Testsome` operations become important for making sure that heartbeat communication completes. This is known as manual progression of MPI messages [39], and plays an important role in the contributions outlined in Chapter 2.

How to process the received times is flexible. The simplest way is to compare the latest time and if one replica exceeds a set tolerance then it is marked as "slow". At this point the faster team could checkpoint it's state at the next heartbeat. Then the single slow rank of team can be swapped out for another. Making a decision based upon a single time interval is likely not a wise idea with the significant overhead of swapping processes. Instead, more advanced statistical techniques may be employed on the entire history of replica timings. For example, exponential smoothing is a commonly used technique used on time series data where older data is considered with less importance. If a process becomes temporarily slow but manages to recover

then the old "slow" times will be outweighed by the recent improved performance. Another approach could be to detect if a process frequently suffers from small slow-downs but then recovers. Such minor slow-downs will not impact the runtime if taken on their own but added together but overall could drastically reduce progress for a team. Clearly, a range of models should be investigated and is a future research topic.

When first testing the single heartbeat idea I discovered a large drawback in its ability to detect a slow *rank*. Although many scientific codes are *iteration* based, there is usually some synchronisation per iteration such as a neighbour communication step or a global collective. The frequency of the synchronisation points depend on the parallelisation properties of the algorithm. Coarse-grained algorithms are able to go longer before synchronising ranks. The consequences for the heartbeat approach is that if one rank is slow, it will also hold back the other members of the same team while they wait for that rank to complete the iteration. With a single heartbeat, it is then impossible to know the offending slow rank, as the whole team will report a slow heartbeat. Therefore I propose a more effective technique, inserting *two* heartbeats into the application code. This time, care must be taken to ensure the two heartbeats are in suitable locations. There must be no synchronisation with other ranks between heartbeat pairs. Although this seems like a large constraint at first, most applications aim to be as coarse-grained as possible. In Section 3.3, I present a trivial application of this idea and then show how it can also be introduced into a highly complex application such as ExaHyPE.

To switch between the single and dual heartbeat modes I use the `sendtag` parameter of the `MPI_Sendrecv` function. A positive tag value starts a heartbeat and a negative tag value ends it. Multiple heartbeats allow teaMPI to measure the times for various sections of code can be introduced with unique tag values. This is a flexible approach. A tag value of zero signals to the library that the single heartbeat algorithm should be used. In the library the data is stored in a map data struc-

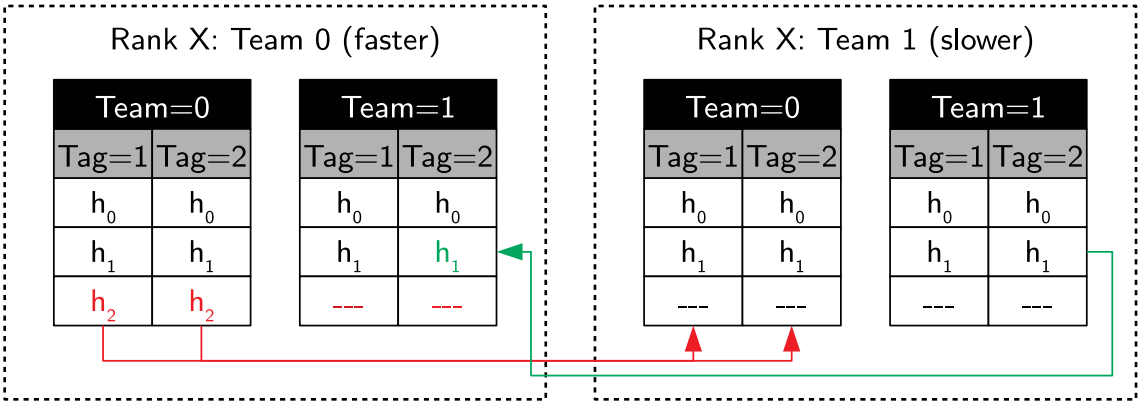


Figure 3.3: An illustration of the heartbeat process with two teams for performance consistency data. The first team is faster than the second. When team 1 triggers the heartbeat for tag 2 of h_1, the data is sent back to team 0 which had already posted the receive. Green indicates a completed communication. Red indicates a pending communication.

ture. A doubly-linked list (`std::list`) of heartbeat times and `MPI_Request` objects (for MPI message progression) are stored per replica, per tag value. I illustrate an example with two teams in Figure 3.3, where team 0 is faster than team 1. Team 1 finally triggers the heartbeat operation h_1 with tag 2. This means the data can be sent back to team 0 where the receive had already been posted. Team 0 has advanced further through the execution and has already triggered both heartbeats for h_2 with a non-blocking send. However, the data will not be picked up by team 1 until it has also reached that point and triggered the receives.

3.2.2 Ensuring Data Consistency

Algorithm 3.5 Compare consistency data between ranks for fault tolerance

```
1: procedure compareConsistency
2:     compareProgress(Algorithm 3.4)
3:     Hash the consistency data with std::hash
4:     for each other team do
5:         Isend hash data to rank in corresponding team
6:         Post Irecv for hash data from rank in corresponding team
7:     MPI_Testsome to check and progress received hashes
8:     if hashes are not equivalent then
9:         Start the recovery process
```

I now detail Algorithm 3.5 which ensures data consistency between replicas. This is important for faults that do not cause a rank to stop (fail-stop) but instead cause it to make incorrect computations. Memory faults are one cause for this. The concept is similar to comparing the times among replicas. The only difference is that the user must supply a data buffer via the `*sendbuf` parameter of `MPI_Sendrecv`. This could be the new solution of an iteration or any characteristic value. Data is then hashed (using `std::hash`) compared among replicas. I neglect the theoretical plausibility of hash collisions in this proof-of-concept. Tags can again be used to discern among different buffer checks alike comparing the times for multiple code sections among replicas. In fact the implementation is almost identical. The values of h in Figure 3.3 now also contain the hashes in addition to the heartbeat times.

If a fault is detected then what happens next depends on how many teams are used. For only two teams then there is no way to correct the error, as we do not know which team is at fault. The best solution here is to roll-back both teams if possible and recompute the data. If one team gets a different result from before then it can be identified as faulty team. If three teams are used, a voting mechanism can tell which team needs to be replaced. It is not usually possible to identify a single rank of a team at fault as the calculation is for most codes contributed by other ranks.

This approach is much more coarse-grained than existing approaches, which check the consistency of all MPI data in every function call. However, I claim that this is excessive, unnecessary and incurs too much overhead. As long as the solution or calculation is consistent per iteration then this implies that all the MPI data in between was also correct. The advantages of such an assumption far outweigh any potential edge-case scenarios. I now discuss how these benefit the overhead of the consistency scheme.

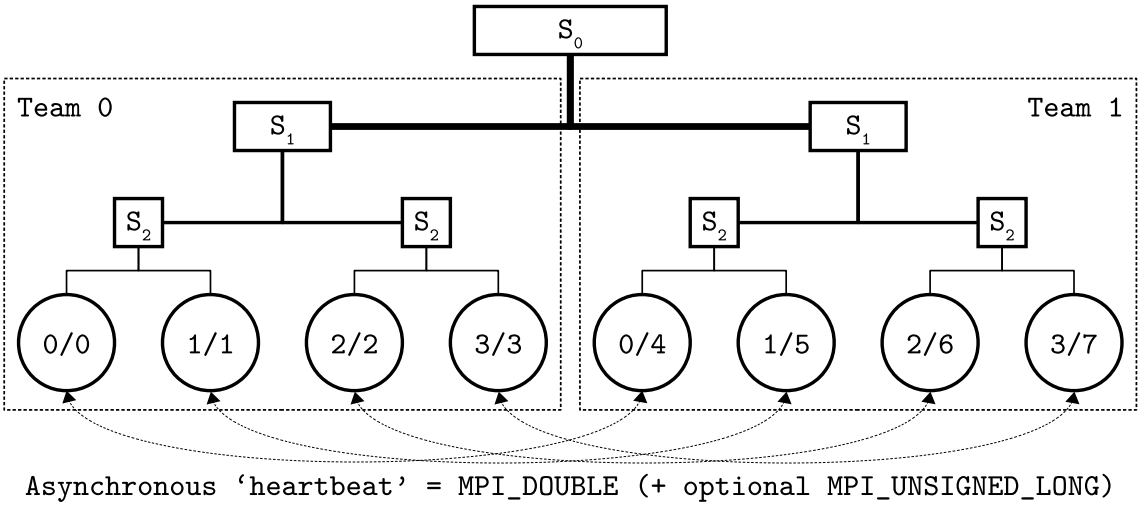


Figure 3.4: An annotated overview of a teaMPI-enabled application with respect to a common 2:1 fat tree network topology. The circles represent ranks, in the form mapped / original rank. The squares represent switches at level $\mathtt{S}_{\mathtt{level}}$, where $\mathtt{S}_0$ represents the root switch.

Overhead Of The Consistency Scheme

Importantly, the teams should be placed as far apart as possible within a cluster. This is the motivation behind the mapping of ranks to team ranks, which by default maps teams to contiguous portion of ranks. The ranks must also be continuously distributed on the native hardware, but this cannot be controlled by teaMPI. If the user wishes the teams to be even further apart, pinning of the ranks can be performed easily by most MPI libraries and batch schedulers.

This logical topology provides two advantages: (i) increased resiliency against faults and (ii) eliminates the extra strain on the network hardware created by replication. The resiliency is increased as it is rare for a fault to spread from a single node, or a clustered set of nodes. Issues predominantly effect either a single node's hardware (e.g. DRAM failure) or hardware supplying a contained set of nodes (e.g. power or network switch failure) [5, 17].

One of the main issues of replication other than the increased hardware demands is that it is assumed that any replication will also have a negative impact on the performance. However, I claim that this not true under a few assumptions.

Table 3.2: Feature comparison of existing rank replication approaches including teaMPI.

	RedMPI	echoMPI	SDR-MPI	teaMPI
Fixed replication	✓	✓	✓	✓
Partial replication	✓	✓		
Platform independent (PMPI)	✓	✓		✓
Error correction	✓	✓		✓
Performance analysis				✓
Work-sharing			✓	
Overhead (1–5)	3/4	5	2	1

For most common network topologies, the intra-team (original application) messages do not overlap. This holds certainly for any tree-based topologies, with a 2:1 fat-tree example given in Figure 3.4. The application is launched on 8 nodes/ranks, and teaMPI creates 2 teams of 4 ranks. Team 0 occupies nodes 0 to 3 and team 1 occupies nodes 4 to 7. Therefore, the bandwidth-heavy application-based network traffic is contained up to switches at level 1 ($\mathtt{S}_1$). Only heartbeat messages cross the top-most switch $\mathtt{S}_0$ and overlap between the two teams. These are small, asynchronous, non-urgent messages issued once per application iteration (infrequently).

The concept can easily be applied to other network topologies too. Provided that the underlying MPI library is also of good quality, the extra processes should also have no impact as all application communication is done via a per-team communicator. Heartbeat messages sent in a dedicated communicator are tiny and therefore non-critical.

In summary, the overhead of the effective consistency scheme here is equal to the overhead of the heartbeat messages, essentially negligible. Variations on the fat tree topology are used everywhere in HPC clusters, from local machines like Hamilton at Durham University to Summit, the world's largest supercomputer to date [56]. They have provably efficient communication for general purpose use [52].

3.2.3 Summary Of Capabilities

With an understanding of the underlying library architecture, I summarise the capabilities based upon the number of teams and heartbeats per iteration:

1. **Two** teams with a **single** heartbeat can identify a slow **team** and provide error **detection**.

2. **Two** teams with **dual** heartbeats can identify a slow **rank** and provide error **detection**.

3. **Three or more** teams with a **single** heartbeat can identify a slow **team** and provide error **correction** via check-pointing.

4. **Three or more** teams with **dual** heartbeats can identify a slow **rank** and provide error **correction** via check-pointing.

In reality the most effective configuration is the second, as the overhead of a dual heartbeat is negligible in comparison to the benefits. Although using three or more ranks does support error correction, the multiplication of resource usage might not justify its gain. It is highly likely that if a error is simply detected, then the application will just be started on a different set of nodes, with the offending nodes replaced. Check-pointing is an expensive operation in both time and resources. I summarise the other capabilities in comparison to the previous approaches from Section 3.1 in Table 3.2. teaMPI omits partial replication as it has been shown to offer limited use for resiliency by James et al [23]. Instead, I offer a library that provides the highest performance, while retaining advanced resiliency and consistency features; all with minimal modification to existing application code.

3.3 Results

In this section I showcase the capabilities of the teaMPI library with experiments based upon three increasingly complex applications:

1. A classical "ping-pong" based acceptance test to investigate the basic bandwidth and latency overhead of teaMPI.

2. A mini application (miniapp) designed to mimic the behaviour of conventional scientific computing applications.

3. A complex ExaHyPE application to show how teaMPI may be used in a real-world setting.

In all three cases, the only changes required were to link against teaMPI (no changes at compile time) and to insert one or two heartbeats with the `MPI_Sendrecv` command as described in Section 3.2. The number of teams is dictated by the `TMPI_TEAMS` environment variable, read when when the application calls `MPI_Init`. The applications are then launched using the chosen MPI library's command. If the usual `mpiexec` command is used then the application is started using `mpiexec -np x*t` where `x` is the ranks per team and `t` is the number of teams. For example, if an application wishes to use three teams of ten ranks then the command would be `mpiexec -np 30`.

The experiments were carried out on the Hamilton 7 cluster at Durham University. Each node consists of 2 x Intel Xeon E5-2659 v4 (Broadwell) 12 core, 2.2 GHz processors. They also have 64 GB of TruDDR4 memory and are connected via an Intel Omnipath 100Gb interconnect in a 2:1 non-blocking fat tree topology.

3.3.1 Ping Pong Test

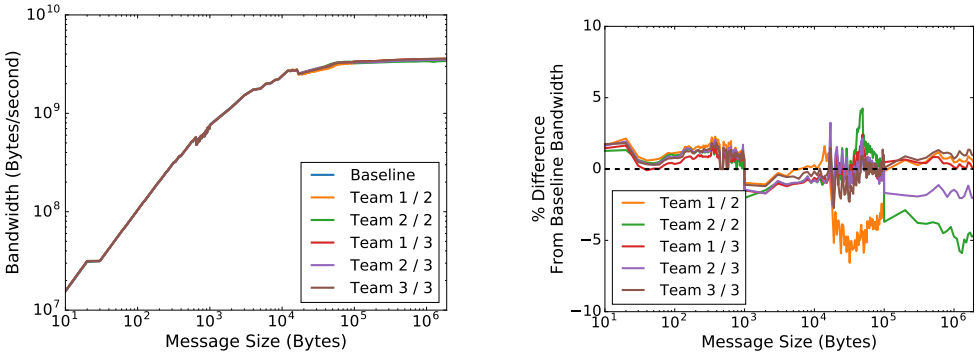
The first experiment to carry out is a simple test to study how much performance is lost by replicating ranks. Previous examples from the literature suggest that it can have a large effect depending on the technique used. With a complete lack of synchronisation among ranks in teaMPI, it is reasonable to expect that the overhead will be negligible. The benchmark chosen is the same as the one used by [26, 29] which measures the bandwidth and latency between ranks in a classic "ping pong"

Algorithm 3.6 Ping Pong acceptance test

```
1: procedure PingPongTest
2:     for t → t_max do
3:         Call MPI_Sendrecv heartbeat
4:         for n_min → n_max do
5:             Start timer
6:             for i → i_max do
7:                 if rank == 0 then
8:                     Send message of size n
9:                     Receive message of size n
10:                else if rank == 1 then
11:                    Receive message of size n
12:                    Send message of size n
13:            Stop timer
14:            new bandwidth ← 2 · i_max/timer
15:            if new bandwidth > bandwidth[n] then
16:                bandwidth[n] ← new bandwidth
```

experiment [35]. The benchmark exchanges increasingly large messages between two ranks starting with only a message envelope as the payload (Algorithm 3.6). As the message size increases, the runtime is dictated by the bandwidth rather than the latency. If teaMPI has any effect on the message passing performance of a rank then it will be shown by an increase in the runtime. Although it is clear that reduced bandwidth will be noticed, a decrease in the latency will also be shown as 10^4 messages are sent back and forth in each trial. 25 trials are performed per message size. The maximum bandwidth over all trials is taken. In this experiment only 2 ranks are required per team, where each team is placed on a separate node of the cluster. This simulates the topology in Figure 3.4 where intra-team communication does not overlap. The `tmi` fabric is used to ensure ranks do not communicate via the specialised shared memory fabric.

The results in Figure 3.5a confirm no performance drop when splitting the original communicator and introducing more teams. Furthermore, the results in Figure 3.5b reinforce the statement that the performance of individual nodes can vary by a large amount. Take for example the experiment with two teams. The first team records a 5% *higher* bandwidth than the baseline of pure MPI for $10^4 < m < 10^5$,



(a) Bandwidth per team

(b) Difference compared to baseline

Figure 3.5: A typical "ping-pong" stress test of the implementation where the bandwidth between two ranks is measured with increasing message size.

where m is the message size in bytes. The second team however records 5% *lower* bandwidth for the same message sizes. The main takeaway from this experiment is that it is valid to claim that teaMPI is a near zero-overhead solution to MPI rank replication, something the previous implementations are not able to. This is through the vast reduction in synchronisation between replicas. I now move onto assessing the capabilities of the heartbeat functionality.

3.3.2 A Typical Scientific Computing Miniapp

Algorithm 3.7 Miniapp

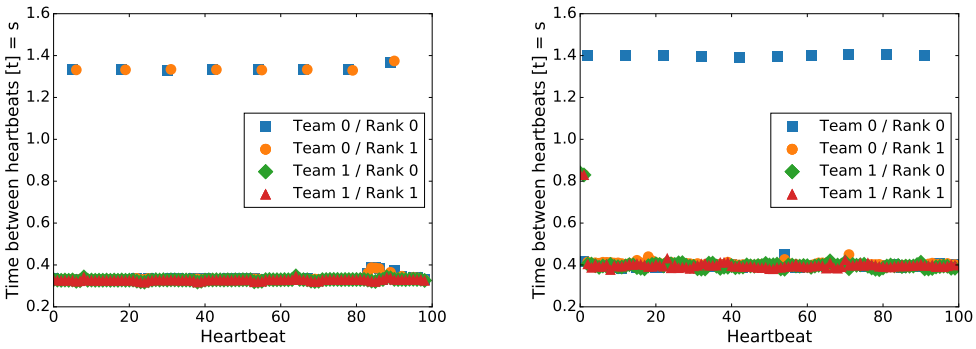
```
1: procedure MINIAPP
2:     for t = t_min → t_max do
3:         MPI_Barrier(MPI_COMM_WORLD)
4:         First heartbeat: MPI_Sendrecv(..., 1, ..., MPI_COMM_SELF)
5:         for i = i_min → i_max do
6:             sin(1.0/3.0)
7:         Second heartbeat: MPI_Sendrecv(..., -1, ..., MPI_COMM_SELF)
8:         MPI_Barrier(MPI_COMM_WORLD)
```

The next experiment demonstrates the capabilities of the teaMPI library with regards to detecting slow teams or ranks. To do so a simple mini application (miniapp) was designed that only performs arbitrary flops in "synchronised" iterations. The simple algorithm is given in Algorithm 3.7. If only one heartbeat is used then the

second `MPI_Sendrecv` is omitted (although it is irrelevant which is removed). This application is useful as it can operate with arbitrary numbers of ranks. It is an excellent proof-of-concept before moving onto real applications in Section 3.3.3. Since it can be safely assumed that the overhead introduced by teaMPI is near-zero, this experiment does not actually perform any communication but instead tests the viability of using heartbeats to detect slow ranks or teams. However, many applications do model this behaviour of iterations with communication followed by computation.

A rank could be slow for a whole range of reasons, such as system load, network load or because it is about to fail. To simulate this the library includes functionality for a benchmark studying the properties of slow ranks, as such behaviour usually occurs non deterministically making it tricky to investigate organically. A signal handler for SIGUSR1 is registered within the library, such that when raised the rank will sleep for one second. The more often this signal is sent to a rank, the slower the progress through the work assigned to it. Therefore it effectively simulates a rank that is slow through more organic means. More sophisticated techniques exist such as lowering the frequency of a CPU [74]. These often require administrator privileges and the sleep command simulates effectively the same behaviour: a slow down of progress. Ultimately, for this test it is not important how the difference in heartbeat times is created.

The first thing I investigated was the use of one versus two heartbeats per iteration (Figure 3.6). For this test the same rank is sent the one second sleep signal every five seconds. This simulates where one rank is much slower than the rest, and the aim is to detect this slow rank. The benchmark based on Algorithm 3.7 was used with 100 iterations, and the time taken to execute 5×10^7 sin operations per iteration was ≈ 0.4 seconds. Figure 3.6a shows the time between heartbeats for when only a single heartbeat per iteration is inserted. Even though only rank 0 in team 0 was sent the sleep signals, the library can only detect that both rank 0 *and* rank 1 in team 0 were slower than their replicas in team 1. This is because rank 1



(a) One heartbeat per iteration (b) Two heartbeats per iteration

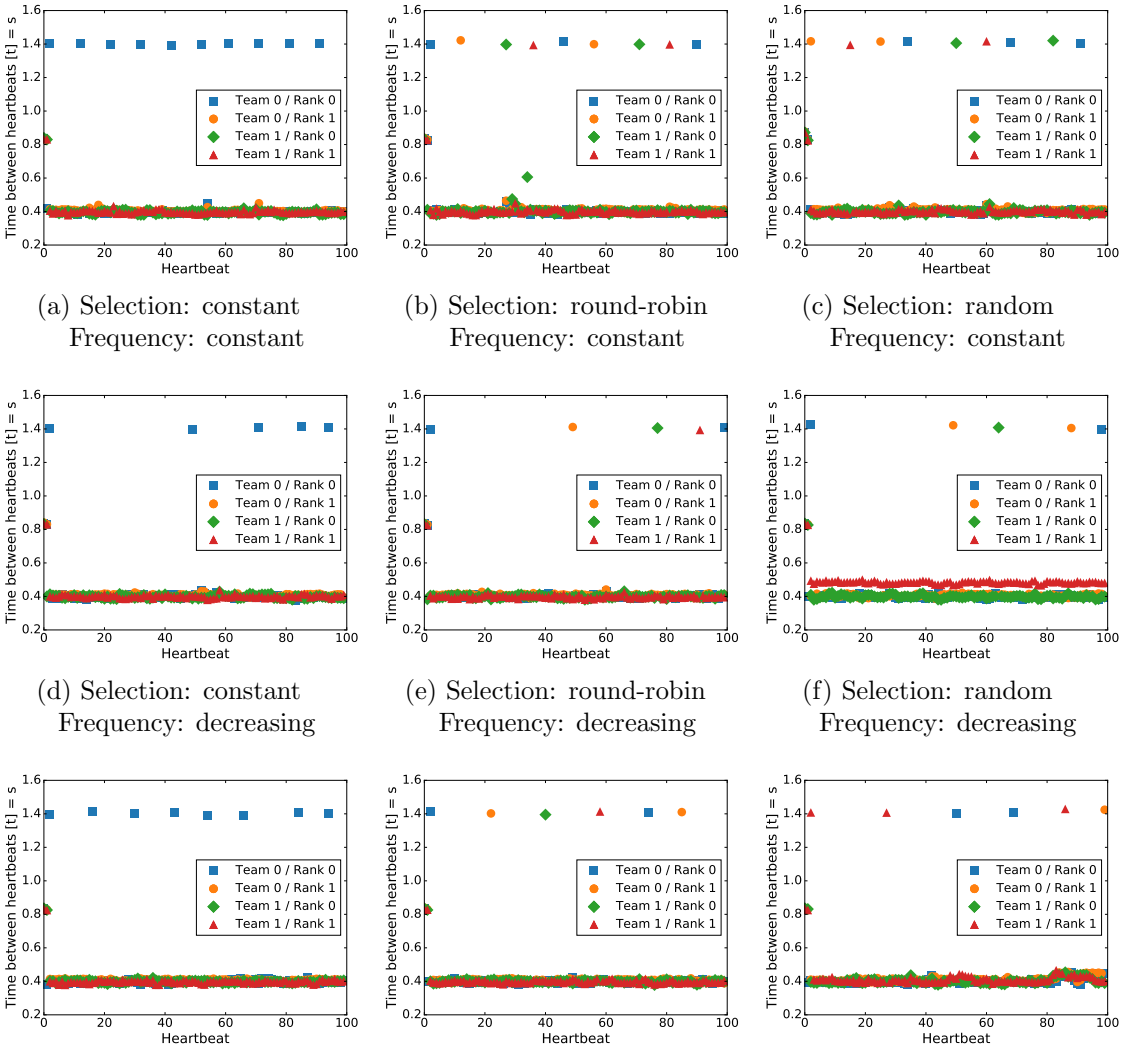
Figure 3.6: An experiment with the benchmark defined in Algorithm 3.7. A single rank is chosen to be "slowed down" with one second sleep commands sent every five seconds. The left is the difference in heartbeat times for a single heartbeat per iteration, where only slow teams can be detected. The right is the difference in heartbeat times for the two heartbeats per iteration. The slow down of rank 0 in team 1 can then be detected individually.

must wait for rank 0 at the `MPI_Barrier` at the end of the iteration. This barrier simulates any kind of synchronisation such as neighbour communication. Therefore, it only hits the heartbeat once rank 0 has caught up. Importantly, teaMPI is able to detect the fault within this team. It took 1 second longer than the other team to compute the same amount of work. In Figure 3.6b, two heartbeats are inserted: one at the start of the work and another at the end as outlined in Algorithm 3.7. This means that there are ranks synchronising between heartbeats. This is clearly visible in the results where this time rank 0 in team 1 was sent the signal every 5 seconds. With two heartbeats teaMPI is able to single out the slow rank which takes 1 second longer than its replica in team 0 to compute the same work. This is a powerful feature, and the future direction of this approach is outlined in Chapter 4.

The second experiment using the miniapp is designed to simulate a wide variety of potential misbehaviour by modifying two variables: to which rank the signal is sent and the frequency of the signals.

1. *Constant*: select the same rank every time

2. *Round robin*: select each rank in turn



(a) Selection: constant
Frequency: constant

(b) Selection: round-robin
Frequency: constant

(c) Selection: random
Frequency: constant

(d) Selection: constant
Frequency: decreasing

(e) Selection: round-robin
Frequency: decreasing

(f) Selection: random
Frequency: decreasing

(g) Selection: constant
Frequency: random

(h) Selection: round-robin
Frequency: random

(i) Selection: random
Frequency: random

Figure 3.7: A series of nine experiments using Algorithm 3.7. Ranks are sent a command to sleep for 1 second with a three different selection criteria and three different frequency intervals. The "selection" and "frequency" parameters are described in Section 3.3.2.

3. *Random*: select rank at random

and the time between signals is one of:

1. *Constant*: always wait a fixed time between signals
2. *Decreasing*: start with a maximum interval between signals then reduce by a fixed factor each time
3. *Random*: wait for a random time between signals

These test cases were chosen as some of the combinations can be envisaged as real world scenarios. Others were included as no one really knows what the runtime behaviour of future hardware will be. However, the teaMPI library is well equipped to monitor any variations between ranks. For example constant selection with a constant interval simulates a slow rank as it will always lag behind the corresponding ranks in other teams. Constant selection and a decreasing interval simulates a failing rank and eventually will be declared dead. Constant selection with a random interval simulates a rank with sporadic performance, maybe being impeded by another application or event. Finally, random selection with a random interval can be seen as an exaggeration of a real HPC environment.

Figure 3.7 shows the results over the whole parameter space. In all cases teaMPI is able to detect the slow-down of individual ranks. I am confident that in real scenarios this statement holds.

3.3.3 A Real Application: LOH.1 Simulation

The next challenge for the implementation was how effective it would work with a "real world" application (Figure 3.8). The chosen set-up is a seismic benchmark simulation known as LOH.1 running on the ExaHyPE engine [73]. The runtime characteristics of the code are highly complex, featuring dynamic asynchronous communication and computation patterns. Adding heartbeats to the code was still trivial.

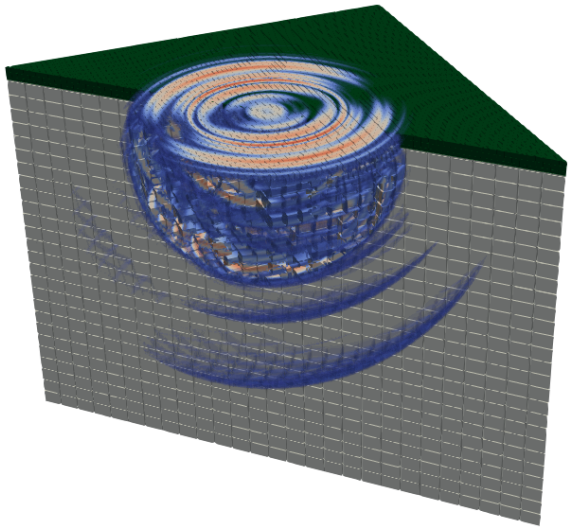


Figure 3.8: Cut through the solution of the LOH.1 benchmark running on the ExaHyPE engine. A point source induces an earthquake just below the surface. Waves propagate from this point but yield complicated patterns as the cubic domain contains two layers of different material [73].

The start beat is still added after the communication phase of the previous timestep and then the end beat is placed before any communication is waited upon.

Chapter 2 thoroughly details the ExaHyPE work-flow but for our replication studies only a knowledge of the phases in Figure 3.9 is required. It begins with start-up phase where the grid is constructed. Then, at the beginning of each timestep the tasks for the timestep are spawned. Here the first heartbeat is placed. A barrier occurs at the end of the timestep where the code waits for the tasks spawned to complete. At the end of this barrier the second heartbeat is triggered. At no point between these heartbeat points does a rank wait for another as the communication barriers occur after the second heartbeat. A final clean-up phase mainly involves deallocating memory. For a simpler presentation of Figure 3.9 the heartbeats are shown aligned but was not the case for the real execution. I reiterate no synchronisation occurs between heartbeats. The communication does not cause any delays in this phase.

The same experimental setup was modified from the previous section to work with the engine. This time 12 ranks per team were used, and the selection criteria

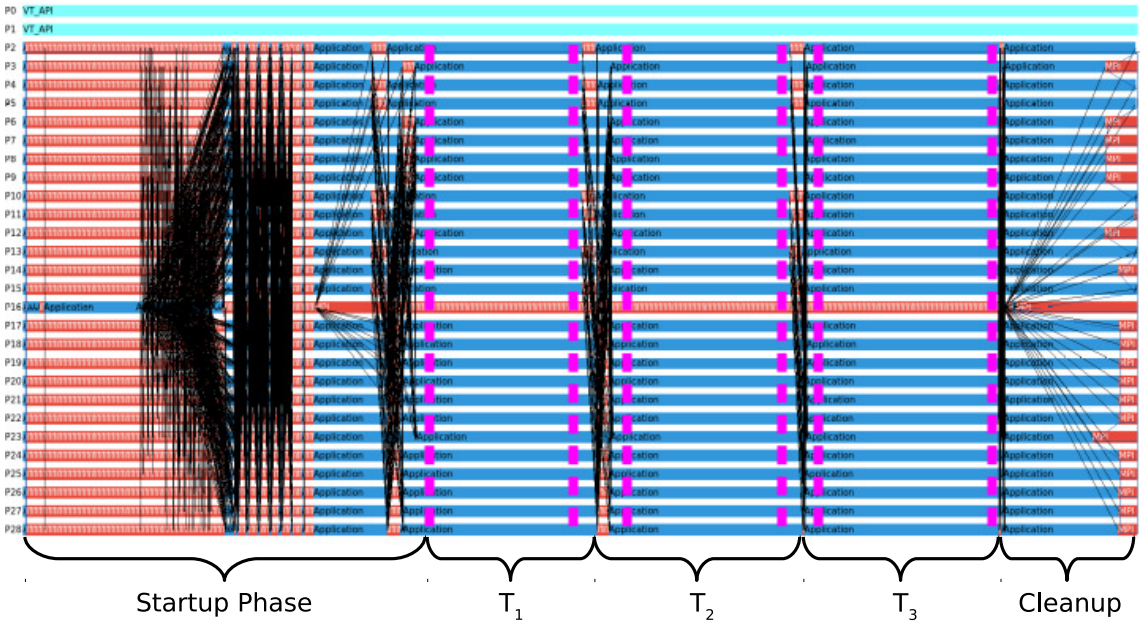
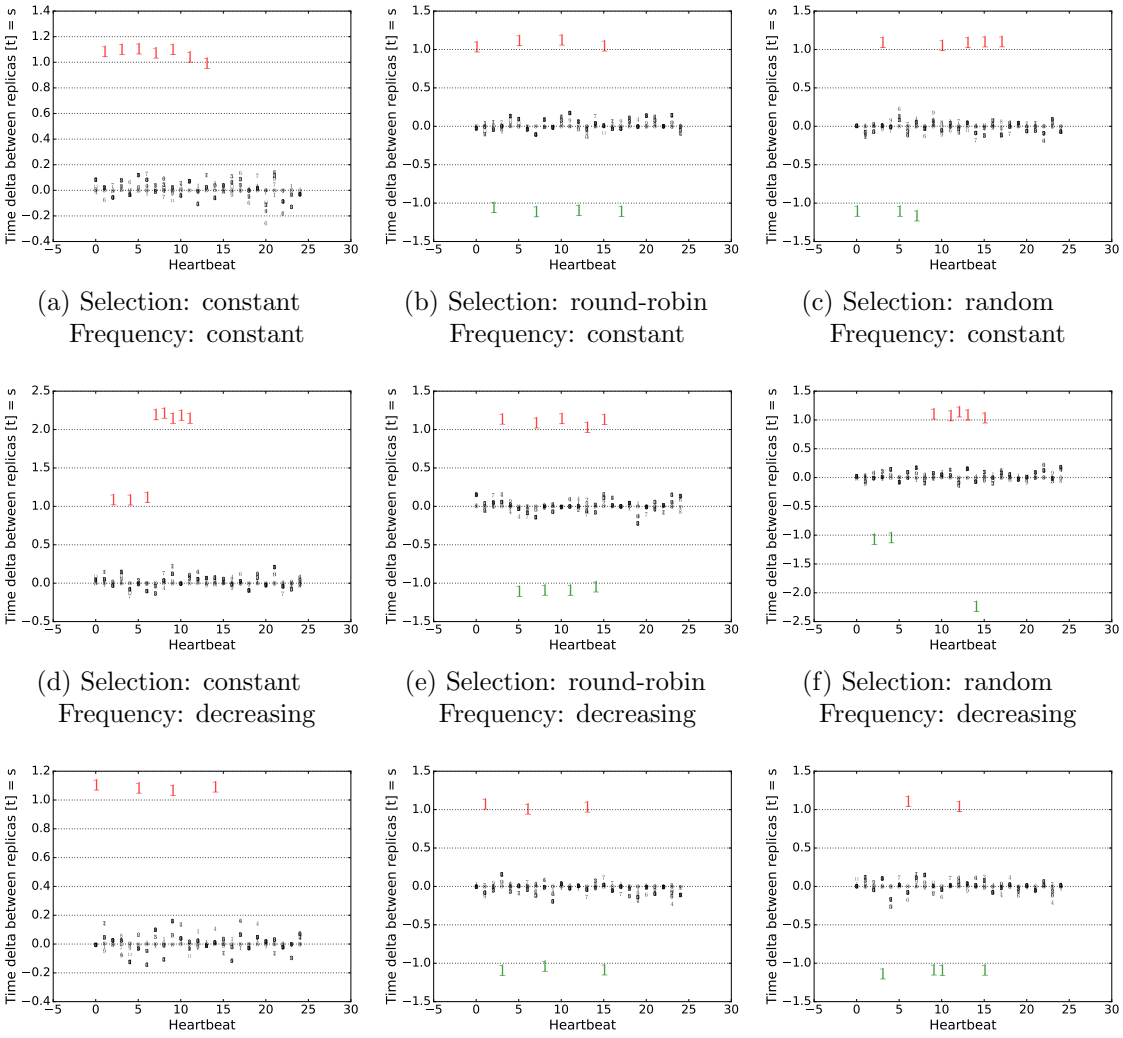


Figure 3.9: A profile of the benchmark code when running for three timesteps on 29 balanced ranks. Example heartbeat locations are marked by the magenta dashed line.

definitions slightly modified. In the initial testing of the experiment I sent the sleep commands to different ranks to simulate a rank slowing down. However, in the engine there is a significant load imbalance in this setup as this is something yet to be tackled (see Chapter 3.4 for some remarks on this). If any other ranks than rank 1 (in any team) are selected to be "slowed-down" via the sleep command then teaMPI will not notice. As rank 1 has the most work to do and the other ranks wait for its results at every timestep after the second heartbeat. Therefore, unless the rank is unrealistically slowed down (or killed) then the sleep command will be called after the work is completed and the end-beat executed. This highlights one potential issue with relying on heartbeats for detecting performance reductions. If such a fault occurs outside of the heartbeats teaMPI cannot detect a slowdown. However, the heartbeat paradigm is designed such that users can maximise the coverage to minimise this risk. The implication is also that if teaMPI does not detect the slowdown, then it is not effecting other ranks. With the proposal to balance tasks among teams more effectively in Chapter 4, the imbalances can be reduced leading



(a) Selection: constant
Frequency: constant

(b) Selection: round-robin
Frequency: constant

(c) Selection: random
Frequency: constant

(d) Selection: constant
Frequency: decreasing

(e) Selection: round-robin
Frequency: decreasing

(f) Selection: random
Frequency: decreasing

(g) Selection: constant
Frequency: random

(h) Selection: round-robin
Frequency: random

(i) Selection: random
Frequency: random

Figure 3.10: The nine experiments from Figure 3.7 repeated with a seismic benchmark application running on the ExaHyPE engine. The "selection" and "frequency" parameters are described in Section 3.3.2. The markers on the scatter plots represent the team rank. The difference in heartbeat times for the first and second replica are plotted. If the first replica is detected to be slow then it is coloured red. If the second replica is detected to be slow then it is coloured green.

to a much improved runtime. To ensure the slowdowns are noticeable by teaMPI, I modify the selection definitions of the benchmark from Section 3.3.2 to be as follows:

1. *Constant*: select rank 1 in the first team every time
2. *Round robin*: select rank 1 but in from a different team in a round-robin fashion each time.
3. *Random*: select rank 1 each time but the team is random.

Additionally, I found that the sleep command interfered with some of the clean-up operations of the code, such as freeing memory. Therefore I only sent the sleep commands within a 45 second window. Clearly, these changes would not be necessary if I had access to more realistic testing methods.

teaMPI is still capable of detecting the various slow-downs of rank 1 (Figure 3.10). What is also noticeable is the increase in variance of the data. The heartbeat times in between vary much more so than the simple benchmark in the previous section. I attribute this to the profile of the code, which has complex characteristics in comparison to the well defined benchmark.

This second experiment showcases the power of teaMPI. With two additional lines of code even in a complex code base such as ExaHyPE, faults in either data or performance can easily be detected and reported by the library. I am confident that this approach can be easily integrated into most scientific codes and possibly even more general applications. Moreover, the addition of the heartbeat messages had no noticeable impact on the performance, validating this lightweight approach to ensuring consistency.

3.4 Outlook

In this chapter I have presented a MPI rank replication scheme. It improves over the numerous existing implementations. My teaMPI library is available as a C++

library that intercepts MPI calls through the dedicated profiling interface. By loosening temporal consistency constraints, replicas are able to operate completely asynchronously from each other. This reduces the performance overhead of replication to practically zero. I validate this claim through the classical "ping-pong" micro-benchmark which showed that increasing the number of replicas has no impact on the bandwidth or latency requirements. At application-specified intervals, called a heartbeat, the replicas exchange data and performance consistency information in a fully non-blocking fashion. This allows the teaMPI library to detect ranks that are slow, failing or producing erroneous results. I presented these capabilities first on a mini-application that simulates the runtime behaviour of many real applications in scientific computing. I show that the integration of teaMPI into a complex code base such as ExaHyPE is trivial, and teaMPI continues to have excellent performance monitoring capabilities of applications with extremely dynamic runtime behaviour properties.

The teaMPI library has several areas for further research. These were not investigated here owing to the scope of the thesis. First, for real fault tolerance the teaMPI library should automatically swap out slow, failing or error producing ranks. Swapping is a complex process that is not supported by many MPI implementations. They exit at the first sign of a failing rank. However, techniques from research into run-through stabilisation techniques show such capabilities are feasible with custom MPI library support [7]. Second, to showcase the capabilities better, real world failure scenarios should be simulated rather than the primitive "sleep" approach in Section 3.3.2/3.3.3. This could be based on existing data provided by many HPC institutions [67, 68]. A final area for future research is the automatic insertion of the heartbeats into the application code. For example, the heartbeats could be inserted between any two synchronising MPI calls. However, with the reduced temporal consistency of teaMPI, an implementation would have to make sure that the heartbeats were called by the "same" functions in all replicas.

In summary, the teaMPI provides a valuable contribution to the existing area of rank replication. If a naïve state machine approach is neglected in favour of the fully asynchronous heartbeat based consistency method proposed in this chapter, advanced fault tolerance techniques can be leveraged transparently to application code with practically zero performance overhead.

In the final chapter of this thesis I bring together the two ideas proposed on tasks and teams. Although powerful concepts in their own right I show they can be combined to equalize work imbalances among individual ranks within a team.

Chapter 4

Conclusion and Synthesis of Contributions

In this thesis I have presented two novel contributions based upon asynchronous algorithms in high performance computing. The first, enclave tasking, uses a producer-consumer idiom and task fusion to ensure high concurrency on dynamically adaptive meshes. As the approach is naturally suited to an overlapping communication approach, it efficiently scales using a hybrid combination of MPI and TBB. It is now used in the ExaHyPE project to use the powerful refinement features of the codebase without suffering the severe overheads of the previous parallel-for based implementation.

The second contribution is the teaMPI library, which creates asynchronous teams from an applications MPI ranks. After a thorough review of existing approaches, I identified a common weakness in that the strong consistency models used induce extremely fine-grained synchronisation among replicas. Therefore by allowing the *application* to decide when to ensure consistency, the overhead is drastically reduced to practically zero. The exchange of both performance and consistency information is implemented in a fully non-blocking fashion and allows the teaMPI library to detect slow or failing ranks and memory corruption errors without the runtime

overhead.

One of the key challenges of the approach outlined in Chapter 2 is ensuring that each rank has an equal amount of tasks to execute. If one has a larger portion of the computational domain then other ranks with less work will have to wait and their computational resources wasted. Optimal domain decompositions with dynamically adaptive grids is a complex topic, as the work per rank may frequently change. This means any decomposition is unlikely to remain valid for long.

I now provide an outlook into a future area of research that combines the two contributions of this thesis into an idea called team-based diffusive load balancing.

The teaMPI library assumes that if replicas are consistent at every heartbeat then they must have executed roughly similar instructions. The final state of all replicas is identical. With TDLB, the consistency model is weakened further such that the replicas no longer operate identically. However, the heartbeat's still mark points where the replicas can compare state.

I outline a use case for this with respect to ExaHyPE. Even if we assume that each cell requires the same amount of time to process, which in the case of non-linear problems is not true, the distribution of work is still non-trivial. If cells are refined or coarsened throughout the simulation then the amount of work on that process will change. This leads to the two challenges to tackle: firstly how to detect the load imbalance detected and secondly how to resolve the imbalance. Existing approaches to the first usually involve abstracting the work into a cost model, which in this case could be the number of cells per rank [65]. In teaMPI we can do better and use the real measurements provided by the heartbeat performance consistency features.

The more challenging, and still an active research area, is how to then rectify the load imbalance? Two main approaches exist. The first requires the application to send and receive cells between ranks as and when required by the load balancer [77]. This approach requires the application to stop and redistribute the work which invokes considerable overhead through synchronisation and data transfer costs. The

second approach allows ranks to steal "tasks" from each other and then send the result back [44, 62–64]. If this happens too often they then do an actual redistribution. However, the issue with both approaches is that data transfer is often considered to be the bottleneck of exascale applications. TDLB is designed to be a data-conservative load balancing scheme.

The core idea of TDLB is for replicas to host *some* cells redundantly, modelling classical overlapping domain-decomposition. However, only a subset replicas will maintain a valid state for a cell. If replicas host the same cells then in ExaHyPE with the contributions from Chapter 2 they will spawn the same tasks into their queues. In the experiments in Section 3.3.3 this means that each replica executes the same STP tasks, an equal amount of work.

However, if the replicas grids do not completely overlap then only some of the STP's are spawned redundantly. With TDLB a heatmap could be embedded into the computational grid. The value in the heatmap would dictate how likely a rank is to provide the result for that cell. Some cells the replica will completely own, and it can process the tasks as normal. Other cells will be owned by multiple replicas, and each send around the result of those cells between each other. On receiving the result of a STP, it can be compared using teaMPI's data consistency features. Throughout the simulation, the overlap will be reduced such that the redundancy in the computation is eliminated.

To summarise, I have presented two novel contributions. In Chapter 2 I investigate a lightweight, asynchronous distributed task system for Discontinuous Galerkin applications. By prioritising critical tasks and efficiently balancing resource usage it promises good scaling even for heavily dynamic adaptive grids up to hundreds of cores. In Chapter 3 I introduce the teaMPI library that replicates MPI ranks to form teams. These teams are able to operate with minimal communication overhead using a novel heartbeat-based consistency scheme. Additional features include the option to efficiently detect data or performance issues for individual ranks. Finally

I show that these two contributions can be brought together with future research surrounding a team based diffusive load balancing scheme. With teams able to share tasks, load can be balanced dynamically at runtime allowing for increasingly complex applications to scale on the worlds largest machines.

Bibliography

[1] G. M. Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities". In: *Proceedings of the April 18-20, 1967, spring joint computer conference*. ACM. 1967, pp. 483–485.

[2] M. Bader et al. *ExaHyPE—an Exascale Hyperbolic PDE solver Engine*. 2017. URL: `http://www.exahype.eu`.

[3] A. Baggag et al. *Parallelization of an object-oriented unstructured aeroacoustics solver*. ICASE Report No. 99-11. 1999.

[4] K. J. Barker et al. "Entering the petaflop era: the architecture and performance of Roadrunner". In: *International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2008, pp. 1–11.

[5] E. Baseman et al. "Physics-Informed Machine Learning for DRAM Error Modeling". In: *The 31st IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems*. 2018.

[6] M. Berger and P. Colella. "Local adaptive mesh refinement for shock hydrodynamics". In: *Journal of Computational Physics* 82 (1989), pp. 64–84.

[7] W. Bland et al. "Post-failure recovery of MPI communication capability: Design and rationale". In: *The International Journal of High Performance Computing Applications* 27.3 (2013), pp. 244–254.

[8] S. Böhm and C. Engelmann. "File I/O for MPI Applications in Redundant Execution Scenarios". In: *Proceedings of the 20th Euromicro International Con-*

ference on Parallel, Distributed, and network-based Processing. IEEE Computer Society, 2012, pp. 112–119.

[9] M. Bougeret et al. "Using group replication for resilience on exascale systems". In: *International Journal of High Performance Computing Applications* 28.2 (2013), pp. 210–224.

[10] F. Cappello. "Fault Tolerance in Petascale/ Exascale Systems: Current Knowledge, Challenges and Research Opportunities". In: *International Journal of High Performance Computing Applications* 23.3 (2009), pp. 212–226.

[11] S. Chakravorty, C. L. Mendes, and L. V. Kalé. "Proactive fault tolerance in MPI applications via task migration". In: *Lecture Notes in Computer Science* 4297 LNCS (2006), pp. 485–496.

[12] D. Charrier and T. Weinzierl. *Stop talking to me—a communication-avoiding ADER-DG realisation*. (submitted). 2018. arXiv: `1801.08682 [cs.MS]`.

[13] D. E. Charrier, B. Hazelwood, and T. Weinzierl. *Enclave Tasking for Discontinuous Galerkin Methods on Dynamically Adaptive Meshes*. 2018. arXiv: `1806.07984 [cs.MS]`.

[14] D. E. Charrier and T. Weinzierl. "An experience report on (auto-) tuning of mesh-based PDE solvers on shared memory systems". In: *International Conference on Parallel Processing and Applied Mathematics*. Springer. 2017, pp. 3–13.

[15] Z. Chen et al. "Fault tolerant high performance computing by a coding approach". In: *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM. 2005, pp. 213–223.

[16] G. Cobb et al. "MPIEcho: A framework for transparent MPI task replication". In: *Dept. of Computer Science, University of Colorado at Boulder, Tech. Rep. CU-CS-1082-11* (2011).

[17] S. Di et al. "Exploring Properties and Correlations of Fatal Events in a Large-Scale HPC System". In: *IEEE Transactions on Parallel and Distributed Systems* (2018).

[18] J. Dongarra et al. "The international exascale software project roadmap". In: *International Journal of High Performance Computing Applications* 25.1 (2011), pp. 3–60.

[19] J. Dongarra et al. *Applied mathematics research for exascale computing*. Tech. rep. Lawrence Livermore National Lab., 2014.

[20] A. Dubey et al. "A Survey of High Level Frameworks in Block-Structured Adaptive Mesh Refinement Packages". In: *CoRR* 74.12 (2016), pp. 3217–3227.

[21] M. Dumbser and M. Käser. "An arbitrary high-order discontinuous Galerkin method for elastic waves on unstructured meshes - II. The three-dimensional isotropic case". In: *Geophysical Journal International* 167.1 (2006), pp. 319–336.

[22] M. Dumbser et al. *Efficient implementation of ADER discontinuous Galerkin schemes for a scalable hyperbolic PDE engine*. 2018. arXiv: `1808.03788 [math.NA]`.

[23] J. Elliott et al. "Combining partial redundancy and checkpointing for HPC". In: *32nd International Conference on Distributed Computing Systems*. IEEE. 2012, pp. 615–626.

[24] C. Engelmann, H. H. Ong, and S. L. Scott. "The case for modular redundancy in large-scale high performance computing systems". In: *Proceedings of the 8th IASTED international conference on parallel and distributed computing and networks* 1 (2009), pp. 189–194.

[25] C. Engelmann. "Scaling to a million cores and beyond: Using light-weight simulation to understand the challenges ahead on the road to exascale". In: *Future Generation Computer Systems* 30 (2014), pp. 59–65.

[26] C. Engelmann and S. Böhm. "Redundant execution of HPC applications with MR-MPI". In: *Proceedings of the 10th IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN)*. 2011, pp. 15–17.

[27] G. E. Fagg et al. "Process fault tolerance: Semantics, design and applications for high performance computing". In: *The International Journal of High Performance Computing Applications* 19.4 (2005), pp. 465–477.

[28] K. Ferreira et al. "Evaluating the viability of process replication reliability for exascale systems". In: *2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2011), pp. 1–12.

[29] K. Ferreira et al. *rMPI: Increasing Fault Resiliency in a Message-Passing Environment*. Tech. rep. Sandia National Laboratories, 2011.

[30] D. Fiala et al. "Detection and Correction of Silent Data Corruption for Large-Scale High-Performance Computing". In: *Proceedings of the 25th IEEE/ACM International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, Nov. 2012, 78:1–78:12.

[31] A. Gainaru, F. Cappello, and W. Kramer. "Taming of the shrew: Modeling the normal and faulty behaviour of large-scale HPC systems". In: *26th International Parallel & Distributed Processing Symposium*. IEEE. 2012, pp. 1168–1179.

[32] A. Gainaru et al. "Fault prediction under the microscope: A closer look into hpc systems". In: *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*. IEEE. 2012, pp. 1–11.

[33] A. Gainaru et al. "Failure prediction for HPC systems and applications: Current situation and open issues". In: *The International Journal of High Performance Computing Applications* 27.3 (2013), pp. 273–282.

[34] T. Gamblin. *wrap (a PMPI wrapper generator)*. https://github.com/LLNL/wrap. 2010.

[35] W. Gropp and E. Lusk. "Reproducible measurements of MPI performance characteristics". In: *European Parallel Virtual Machine/Message Passing Interface User's Group Meeting*. Springer. 1999, pp. 11–18.

[36] J. L. Gustafson. "Reevaluating Amdahl's law". In: *Communications of the ACM* 31.5 (1988), pp. 532–533.

[37] G. Hager and G. Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. 1st. Boca Raton, FL, USA: CRC Press, Inc., 2010.

[38] B. Hazelwood. *teaMPI: a team based PMPI wrapper for MPI resiliency*. 2018. URL: `http://www.peano-framework.org/hpcsoftware/teampi`.

[39] T. Hoefler and A. Lumsdaine. "Message progression in parallel computing - to thread or not to thread?" In: *IEEE International Conference on Cluster Computing*. 2008, pp. 213–222.

[40] M. Hutchinson et al. "Efficiency of High Order Spectral Element Methods on Petascale Architectures". In: *High Performance Computing*. Ed. by J. M. Kunkel, P. Balaji, and J. Dongarra. Cham: Springer International Publishing, 2016, pp. 449–466.

[41] A. Ilic, F. Pratas, and L. Sousa. "Cache-aware Roofline model: Upgrading the loft". In: *IEEE Computer Architecture Letters* 13.1 (2014), pp. 21–24.

[42] Intel. *Product Change Notification 116378 - 00*. URL: `http://qdms.intel.com/dm/i.aspx/9C54A9A7-BF37-4496-B268-BD2746EA54D3/PCN116378-00.pdf`.

[43] Intel. *Intel Threading Building Blocks Design Patterns*. 2010. URL: `https://software.intel.com/sites/default/files/m/4/8/1/e/e/33963-Design_Patterns.pdf`.

[44] L. V. Kale and G. Zheng. "Charm++ and AMPI: Adaptive Runtime Strategies via Migratable Objects". In: *Advanced Computational Infrastructures for Parallel and Distributed Applications*. Ed. by M. Parashar. Wiley-Interscience, 2009, pp. 265–282.

[45] G. Karypis and V. Kumar. "A fast and high quality multilevel scheme for partitioning irregular graphs". In: *SIAM Journal on scientific Computing* 20.1 (1998), pp. 359–392.

[46] T. Kolda. *What Kind of Science Is Computational Science? A Rebuttal*. 2014. URL: `https://sinews.siam.org/Details-Page/what-kind-of-science-is-computational-science-a-rebuttal`.

[47] D. Komatitsch et al. "High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster". In: *Journal of Computational Physics* 229 (2010), pp. 7692–7714.

[48] K. Kormann and M. Kronbichler. "Parallel finite element operator application: Graph partitioning and coloring". In: *7th International Conference on E-Science*. IEEE. 2011, pp. 332–339.

[49] M. Kronbichler and K. Kormann. *Fast matrix-free evaluation of discontinuous Galerkin finite element operators*. 2017. arXiv: `1711.03590 [cs.MS]`.

[50] M. Kronbichler et al. "Fast Matrix-Free Discontinuous Galerkin Kernels on Modern Computer Architectures". In: *High Performance Computing*. Springer International Publishing, 2017, pp. 237–255.

[51] A. Lefray, T. Ropars, and A. Schiper. "Replication for send-deterministic MPI HPC applications". In: *Proceedings of the 3rd Workshop on Fault-tolerance for HPC at extreme scale*. ACM. 2013, pp. 33–40.

[52] C. E. Leiserson. "Fat-trees: universal networks for hardware-efficient supercomputing". In: *IEEE transactions on Computers* 100.10 (1985), pp. 892–901.

[53] R. J. LeVeque. *Finite-Volume Methods for Hyperbolic Problems*. Cambridge University Press, 2002.

[54] G. E. Moore. "Cramming more components onto integrated circuits," in: *Electronics* (1965).

[55] MPI Forum. *MPI: A Message Passing Interface Standard Version 3.1*. 2015. URL: `https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf`.

[56] Oak Ridge National Laboratory. *Genomics Code Exceeds Exaops On Summit Supercomputer*. 2018. URL: `https://www.olcf.ornl.gov/2018/06/08/genomics-code-exceeds-exaops-on-summit-supercomputer/`.

[57] A. Pop and A. Cohen. "A stream-computing extension to OpenMP". In: *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*. ACM. 2011, pp. 5–14.

[58] J. Reinders. *Intel Threading Building Blocks*. First. O'Reilly & Associates, Inc., 2007.

[59] R. Riesen, K. Ferreira, and J. Stearley. "See applications run and throughput jump: The case for redundant computing in HPC". In: *Proceedings of the International Conference on Dependable Systems and Networks* (2010), pp. 29–34.

[60] T. Ropars et al. "Efficient Process Replication for MPI Applications: Sharing Work Between Replicas". In: *Parallel and Distributed Processing Symposium*. IEEE. 2015, pp. 645–654.

[61] B. Rountree et al. "Parallelizing heavyweight debugging tools with MPIEcho". In: *Parallel Computing* 39.3 (2013), pp. 156–166.

[62] P. Samfass. *Towards Reactive Task-Based Work Stealing in Distributed Memory in sam(oa)*^{2}. TU Darmstadt: Technical University of Munich, Sept. 2017.

[63] P. Samfass. *Parallel Adaptive Mesh Refinement in Sam(oa)*^{2} *- Load Balancing vs. Work Stealing*. Tokio: Waseda University, Mar. 2018.

[64] P. Samfass, J. Klinkenberg, and M. Bader. "Hybrid MPI+OpenMP reactive work stealing in distributed memory in the PDE framework sam(oa)^2". In: *IEE International Conference on Cluster Computing*. Accepted. 2018.

[65] M. Schaller et al. "SWIFT: Using task-based parallelism, fully asynchronous communication, and graph partition-based domain decomposition for strong scaling on more than 100,000 cores". In: *Proceedings of the Platform for Advanced Scientific Computing Conference*. ACM. 2016, p. 2.

[66] M. Schreiber, T. Weinzierl, and H. J. Bungartz. "Cluster Optimization and Parallelization of Simulations with Dynamically Adaptive Grids". In: *Euro-Par 2013 Parallel Processing*. Vol. 8097. Lecture Notes in Computer Science. Springer, 2013, pp. 484–496.

[67] B. Schroeder and G. A. Gibson. "Understanding failures in petascale computers". In: *Journal of Physics: Conference Series* 78 (2007), p. 012022.

[68] B. Schroeder and G. a. Gibson. "A Large-Scale Study of Failures in High-Performance Computing Systems". In: *IEEE Transactions on Dependable and Secure Computing* 7.4 (2010), pp. 337–350.

[69] M. Schulz and B. R. De Supinski. "A flexible and dynamic infrastructure for MPI tool interoperability". In: *International Conference on Parallel Processing*. IEEE. 2006, pp. 193–202.

[70] M. Schulz and B. R. De Supinski. "P^{N}MPI tools: A whole lot greater than the sum of their parts". In: *Proceedings of the ACM/IEEE conference on Supercomputing*. ACM. 2007, p. 30.

[71] H. Sundar and O. Ghattas. "A Nested Partitioning Algorithm for Adaptive Meshes on Heterogeneous Clusters". In: *Proceedings of the 29th ACM on International Conference on Supercomputing*. ICS '15. ACM, 2015, pp. 319–328.

[72] M. Tavelli et al. *A simple diffuse interface approach on adaptive Cartesian grids for the linear elastic wave equations with complex topography*. 2018. arXiv: `1804.09491 [math.NA]`.

[73] The SPICE Code Validation. *Problem WP2_LOH1*. 2006. URL: `http://www.sismowine.org/model/WP2_LOH1.pdf`.

[74] J. Treibig, G. Hager, and G. Wellein. "LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments". In: *Proceedings of the 39th International Conference on Parallel Processing Workshops*. IEEE Computer Society, 2010, pp. 207–216.

[75] C. Uphoff et al. "Extreme Scale Multi-physics Simulations of the Tsunamigenic 2004 Sumatra Megathrust Earthquake". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2017.

[76] C. Wang et al. "Proactive process-level live migration in HPC environments". In: *Proceedings of the ACM/IEEE conference on Supercomputing*. IEEE Press. 2008, p. 43.

[77] T. Weinzierl. *The Peano software - parallel, automaton-based, dynamically adaptive grid traversals*. 2018. arXiv: `1506.04496v5 [cs.MS]`.

[78] S. Williams, A. Waterman, and D. Patterson. "Roofline: an insightful visual performance model for multicore architectures". In: *Communications of the ACM* 52.4 (2009), pp. 65–76.

[79] M. Wittmann et al. *Asynchronous MPI for the Masses*. 2013. arXiv: `1302.4280 [cs.DC]`.

[80] Z. Zheng and Z. Lan. "Reliability-Aware Scalability Models for High Performance Computing". In: *IEEE International Conference on Cluster Computing and Workshops* (2009), pp. 1–9.

[81] Z. Zheng et al. “A practical failure prediction with location and lead time for blue gene/p”. In: *International Conference on Dependable Systems and Networks Workshops*. IEEE. 2010, pp. 15–22.